

Quantum and Classical Registers

Dominique Unruh

December 14, 2021

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	2
2	Generic laws about registers	3
2.1	Elementary facts	4
2.2	Preregisters	4
2.3	Registers	4
2.4	Tensor product of registers	4
2.5	Pairs and compatibility	6
2.6	Fst and Snd	8
2.7	Compatibility of register tensor products	10
2.8	Associativity of the tensor product	11
2.9	Iso-registers	12
2.10	Compatibility simplification	15
2.11	Notation	16
3	Axioms of complements	16
4	Generic laws about complements	17
5	Classical instantiation of registers	23
6	Generic laws about registers, instantiated classically	27
6.1	Elementary facts	27
6.2	Preregisters	28
6.3	Registers	28
6.4	Tensor product of registers	28
6.5	Pairs and compatibility	30
6.6	Fst and Snd	32

6.7	Compatibility of register tensor products	34
6.8	Associativity of the tensor product	35
6.9	Iso-registers	36
6.10	Compatibility simplification	39
6.11	Notation	40
7	Miscellaneous facts	40
8	Derived facts about classical registers	43
9	Tensor products (finite dimensional)	45
10	Quantum instantiation of registers	58
11	Generic laws about registers, instantiated quantumly	60
11.1	Elementary facts	60
11.2	Preregisters	60
11.3	Registers	61
11.4	Tensor product of registers	61
11.5	Pairs and compatibility	63
11.6	Fst and Snd	65
11.7	Compatibility of register tensor products	67
11.8	Associativity of the tensor product	68
11.9	Iso-registers	69
11.10	Compatibility simplification	71
11.11	Notation	73
12	Quantum mechanics basics	73
12.1	Basic quantum states	73
12.1.1	EPR pair	73
12.1.2	Ket plus	73
12.2	Basic quantum gates	73
12.2.1	Pauli X	73
12.2.2	Pauli Z	74
12.2.3	Hadamard	74
12.2.4	CNOT	74
12.2.5	Qubit swap	75
13	Derived facts about quantum registers	75
14	Very simple Quantum Hoare logic	78
15	Tensor products as matrices	79
16	Quantum teleportation	83
17	Quantum instantiation of complements	87
18	Generic laws about complements, instantiated quantumly	98
19	More derived facts about quantum registers	104

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 \Rightarrow 'a update \Rightarrow '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 \Rightarrow 'b::domain update⟩ \Rightarrow bool

axiomatization **where**

comp-preregister: *preregister* F \Longrightarrow *preregister* G \Longrightarrow *preregister* (G \circ F) **and**

id-preregister: ⟨*preregister* *id*⟩

for F :: ⟨'a::domain update \Rightarrow 'b::domain update⟩ **and** G :: ⟨'b update \Rightarrow '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 \Rightarrow 'b::domain update \Rightarrow ('a \times 'b) update⟩

where *tensor-extensionality*: *preregister* F \Longrightarrow *preregister* G \Longrightarrow (\bigwedge a b. F (*tensor-update* a b) = G (*tensor-update* a b)) \Longrightarrow F = G

for F G :: ⟨('a \times 'b) update \Rightarrow '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 \Rightarrow 'b update) \Rightarrow bool⟩

axiomatization **where**

register-preregister: *register* F \Longrightarrow *preregister* F **and**

register-comp: *register* F \Longrightarrow *register* G \Longrightarrow *register* (G \circ F) **and**

register-mult: *register* F \Longrightarrow *comp-update* (F a) (F b) = F (*comp-update* a b) **and**

register-of-id: ⟨*register* F \Longrightarrow F *id-update* = *id-update*⟩ **and**

register-id: ⟨*register* (*id* :: 'a update \Rightarrow 'a update)⟩

for F :: 'a::domain update \Rightarrow 'b::domain update **and** G :: 'b update \Rightarrow '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 \Rightarrow 'c::domain update⟩ \Rightarrow ('b::domain update \Rightarrow 'c update)

\Rightarrow ((('a \times 'b) update \Rightarrow 'c update) **where**

register-pair-is-register: ⟨*register* F \Longrightarrow *register* G \Longrightarrow (\bigwedge a b. *comp-update* (F a) (G b) = *comp-update* (G b) (F a))

\Longrightarrow *register* (*register-pair* F G) **and**

register-pair-apply: ⟨*register* F \Longrightarrow *register* G \Longrightarrow (\bigwedge a b. *comp-update* (F a) (G b) = *comp-update* (G b) (F a))

\Longrightarrow (*register-pair* F G) (*tensor-update* a b) = *comp-update* (F a) (G b)⟩

end

2 Generic laws about registers

theory *Laws*

imports *Axioms*

begin

This notation is only used inside this file

notation *comp-update* (**infixl** *_u 55)

notation *tensor-update* (**infixr** \otimes_u 70)

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

2.1 Elementary facts

```

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

```

2.2 Preregisters

```

lemma preregister-tensor-left[simp]: ⟨preregister (λb::'b::domain update. tensor-update a b)⟩
  for a :: ⟨'a::domain update⟩
proof -
  have ⟨preregister ((λb1::('a×'b) update. (a ⊗u id-update) *u b1) o (λb. tensor-update id-update b))⟩
    by (rule comp-preregister; simp)
  then show ?thesis
    by (simp add: o-def tensor-update-mult)
qed

```

```

lemma preregister-tensor-right[simp]: ⟨preregister (λa::'a::domain update. tensor-update a b)⟩
  for b :: ⟨'b::domain update⟩
proof -
  have ⟨preregister ((λa1::('a×'b) update. (id-update ⊗u b) *u a1) o (λa. tensor-update a id-update))⟩
    by (rule comp-preregister, simp-all)
  then show ?thesis
    by (simp add: o-def tensor-update-mult)
qed

```

2.3 Registers

```

lemma id-update-tensor-register[simp]:
  assumes ⟨register F⟩
  shows ⟨register (λa::'a::domain update. id-update ⊗u F a)⟩
  using assms apply (rule register-comp[unfolded o-def])
  by simp

```

```

lemma register-tensor-id-update[simp]:
  assumes ⟨register F⟩
  shows ⟨register (λa::'a::domain update. F a ⊗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 (λa. tensor-update (F a) id-update) (λb. 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* ($-::'b::\text{domain itself}$) $A \longleftrightarrow$
 $(\forall F G :: 'a::\text{domain update} \Rightarrow 'b \text{ update. preregister } F \longrightarrow \text{preregister } G \longrightarrow (\forall x \in A. F x = G x) \longrightarrow F = G)$

lemma *separating-UNIV*[*simp*]: $\langle \text{separating TYPE}(-) \text{ UNIV} \rangle$
unfolding *separating-def* **by** *auto*

lemma *separating-mono*: $\langle A \subseteq B \Longrightarrow \text{separating TYPE}('a::\text{domain}) A \Longrightarrow \text{separating TYPE}('a) B \rangle$
unfolding *separating-def* **by** (*meson in-mono*)

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

lemma *separating-tensor*:

fixes $A :: \langle 'a::\text{domain update set} \rangle$ **and** $B :: \langle 'b::\text{domain update set} \rangle$
assumes [*simp*]: $\langle \text{separating TYPE}('c::\text{domain}) A \rangle$
assumes [*simp*]: $\langle \text{separating TYPE}('c) B \rangle$
shows $\langle \text{separating TYPE}('c) \{a \otimes_u b \mid a b. a \in A \wedge b \in B\} \rangle$
proof (*unfold separating-def, intro allI impI*)
fix $F G :: \langle ('a \times 'b) \text{ update} \Rightarrow 'c \text{ update} \rangle$
assume [*simp*]: $\langle \text{preregister } F \rangle \langle \text{preregister } G \rangle$
have [*simp*]: $\langle \text{preregister } (\lambda x. F (a \otimes_u x)) \rangle$ **for** a
using - $\langle \text{preregister } F \rangle$ **apply** (*rule comp-preregister[unfolded o-def]*)
by *simp*
have [*simp*]: $\langle \text{preregister } (\lambda x. G (a \otimes_u x)) \rangle$ **for** a
using - $\langle \text{preregister } G \rangle$ **apply** (*rule comp-preregister[unfolded o-def]*)
by *simp*
have [*simp*]: $\langle \text{preregister } (\lambda x. F (x \otimes_u b)) \rangle$ **for** b
using - $\langle \text{preregister } F \rangle$ **apply** (*rule comp-preregister[unfolded o-def]*)
by *simp*
have [*simp*]: $\langle \text{preregister } (\lambda x. G (x \otimes_u b)) \rangle$ **for** b
using - $\langle \text{preregister } G \rangle$ **apply** (*rule comp-preregister[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 $\langle EQ: \langle F (a \otimes_u b) = G (a \otimes_u b) \rangle$ **if** $\langle a \in A \rangle$ **and** $\langle b \in B \rangle$ **for** $a b$
using *that* **by** *auto*
then have $\langle F (a \otimes_u b) = G (a \otimes_u b) \rangle$ **if** $\langle a \in A \rangle$ **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*]: $\langle \text{register } F \rangle \langle \text{register } G \rangle \langle \text{register } H \rangle \langle \text{register } L \rangle$
shows $\langle (F \otimes_r G) \circ (H \otimes_r L) = (F \circ H) \otimes_r (G \circ L) \rangle$
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 $A :: \langle 'a::\text{domain update set} \rangle$ **and** $B :: \langle 'b::\text{domain update set} \rangle$
assumes $\langle \text{separating TYPE}('c::\text{domain}) A \rangle$

assumes $\langle \text{separating TYPE}(c) B \rangle$
assumes $\langle C = \{a \otimes_u b \mid a b. a \in A \wedge b \in B\} \rangle$
shows $\langle \text{separating TYPE}(c) C \rangle$
using *assms*
by (*simp add: separating-tensor*)

lemma *tensor-extensionality3*:

fixes $F G :: \langle ('a::\text{domain} \times 'b::\text{domain} \times 'c::\text{domain}) \text{ update} \Rightarrow 'd::\text{domain} \text{ update} \rangle$
assumes [*simp*]: $\langle \text{register } F \rangle \langle \text{register } G \rangle$
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** $A = \{a \otimes_u b \otimes_u c \mid a b c. \text{True}\}$]])

have $\langle \text{separating TYPE}(d) \{b \otimes_u c \mid b c. \text{True}\} \rangle$

apply (*rule separating-tensor'*[**where** $A = \text{UNIV}$ **and** $B = \text{UNIV}$]])

by *auto*

then show $\langle \text{separating TYPE}(d) \{a \otimes_u b \otimes_u c \mid a b c. \text{True}\} \rangle$

apply (*rule-tac separating-tensor'*[**where** $A = \text{UNIV}$ **and** $B = \{b \otimes_u c \mid b c. \text{True}\}$]])

by *auto*

show $\langle \text{preregister } F \rangle \langle \text{preregister } G \rangle$ **by** *auto*

show $\langle x \in \{a \otimes_u b \otimes_u c \mid a b c. \text{True}\} \Longrightarrow F x = G x \rangle$ **for** x

using *assms(3)* **by** *auto*

qed

lemma *tensor-extensionality3'*:

fixes $F G :: \langle ('a::\text{domain} \times 'b::\text{domain}) \times 'c::\text{domain} \rangle \text{ update} \Rightarrow 'd::\text{domain} \text{ update} \rangle$
assumes [*simp*]: $\langle \text{register } F \rangle \langle \text{register } G \rangle$
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** $A = \{(a \otimes_u b) \otimes_u c \mid a b c. \text{True}\}$]])

have $\langle \text{separating TYPE}(d) \{a \otimes_u b \mid a b. \text{True}\} \rangle$

apply (*rule separating-tensor'*[**where** $A = \text{UNIV}$ **and** $B = \text{UNIV}$]])

by *auto*

then show $\langle \text{separating TYPE}(d) \{(a \otimes_u b) \otimes_u c \mid a b c. \text{True}\} \rangle$

apply (*rule-tac separating-tensor'*[**where** $B = \text{UNIV}$ **and** $A = \{a \otimes_u b \mid a b. \text{True}\}$]])

by *auto*

show $\langle \text{preregister } F \rangle \langle \text{preregister } G \rangle$ **by** *auto*

show $\langle x \in \{(a \otimes_u b) \otimes_u c \mid a b c. \text{True}\} \Longrightarrow F x = G x \rangle$ **for** x

using *assms(3)* **by** *auto*

qed

lemma *register-tensor-id*[*simp*]: $\langle \text{id} \otimes_r \text{id} = \text{id} \rangle$

apply (*rule tensor-extensionality*)

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

2.5 Pairs and compatibility

definition *compatible* :: $\langle ('a::\text{domain} \text{ update} \Rightarrow 'c::\text{domain} \text{ update})$

$\Rightarrow ('b::\text{domain} \text{ update} \Rightarrow 'c \text{ update}) \Rightarrow \text{bool} \rangle$ **where**

$\langle \text{compatible } F G \iff \text{register } F \wedge \text{register } G \wedge (\forall a b. F a *_u G b = G b *_u F a) \rangle$

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* $x y \Longrightarrow \text{compatible } y x$

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 $\langle \text{compatible } F \ G \rangle$
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 $\langle \text{compatible } F \ G \rangle$
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 \implies register H \implies compatible (F o H) G*
by (*simp add: compatible-def*)

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

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

lemma *compatible-register1*: $\langle \text{compatible } F \ G \implies \text{register } F \rangle$
by (*simp add: compatible-def*)

lemma *compatible-register2*: $\langle \text{compatible } F \ G \implies \text{register } G \rangle$
by (*simp add: compatible-def*)

lemma *pair-o-tensor*:
assumes *compatible A B* **and** [simp]: $\langle \text{register } C \rangle$ **and** [simp]: $\langle \text{register } D \rangle$
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 $F :: 'a::\text{domain update} \Rightarrow 'c::\text{domain update}$ **and** $G :: 'b::\text{domain update} \Rightarrow 'c::\text{domain update}$
assumes *compatible F G*
shows *compatible* $(\lambda a. \text{id-update} \otimes_u F a) (\lambda a. \text{id-update} \otimes_u G a)$
using *assms apply (rule compatible-comp-inner[unfolded o-def])*
by *simp*

lemma *compatible-tensor-id-update-right*[simp]:
fixes $F :: 'a::\text{domain update} \Rightarrow 'c::\text{domain update}$ **and** $G :: 'b::\text{domain update} \Rightarrow 'c::\text{domain update}$
assumes *compatible F G*
shows *compatible* $(\lambda a. F a \otimes_u \text{id-update}) (\lambda a. G a \otimes_u \text{id-update})$
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* $(\lambda a. F a \otimes_u \text{id-update}) (\lambda a. \text{id-update} \otimes_u G a)$
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* $(\lambda a. \text{id-update } \otimes_u F a) (\lambda a. G a \otimes_u \text{id-update})$
apply (*rule compatibleI*)
using *assms* **by** (*auto simp: tensor-update-mult*)

lemma *register-comp-pair*:

assumes [*simp*]: $\langle \text{register } F \rangle$ **and** [*simp*]: $\langle \text{compatible } G H \rangle$
shows $(F \circ G; F \circ H) = F \circ (G; H)$

proof (*rule tensor-extensionality*)

show $\langle \text{preregister } (F \circ G; F \circ H) \rangle$ **and** $\langle \text{preregister } (F \circ (G; H)) \rangle$
by *simp-all*

have [*simp*]: $\langle \text{compatible } (F \circ G) (F \circ H) \rangle$
apply (*rule compatible-comp-inner, simp*)
by *simp*

then have [*simp*]: $\langle \text{register } (F \circ G) \rangle$ $\langle \text{register } (F \circ H) \rangle$

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

from *assms* **have** [*simp*]: $\langle \text{register } G \rangle$ $\langle \text{register } H \rangle$

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

fix *a b*

show $\langle (F \circ G; F \circ H) (a \otimes_u b) = (F \circ (G; H)) (a \otimes_u b) \rangle$

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** $\langle \text{Fst } a = a \otimes_u \text{id-update} \rangle$

definition *Snd* **where** $\langle \text{Snd } a = \text{id-update } \otimes_u a \rangle$

lemma *register-Fst*[*simp*]: $\langle \text{register } \text{Fst} \rangle$

unfolding *Fst-def* **by** (*rule register-tensor-left*)

lemma *register-Snd*[*simp*]: $\langle \text{register } \text{Snd} \rangle$

unfolding *Snd-def* **by** (*rule register-tensor-right*)

lemma *compatible-Fst-Snd*[*simp*]: $\langle \text{compatible } \text{Fst } \text{Snd} \rangle$

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 $\langle \text{swap} = (\text{Snd}; \text{Fst}) \rangle$

lemma *swap-apply*[*simp*]: $\text{swap } (a \otimes_u b) = (b \otimes_u a)$

unfolding *swap-def*

by (*simp add: Axioms.register-pair-apply Fst-def Snd-def tensor-update-mult*)

lemma *swap-o-Fst*: $\text{swap } o \text{Fst} = \text{Snd}$

by (*auto simp add: Fst-def Snd-def*)

lemma *swap-o-Snd*: $\text{swap } o \text{Snd} = \text{Fst}$

by (*auto simp add: Fst-def Snd-def*)

lemma *register-swap*[simp]: $\langle \text{register swap} \rangle$
by (*simp add: swap-def*)

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

lemma *swap-o-swap*[simp]: $\langle \text{swap o swap} = id \rangle$
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]: $\langle \text{swap (swap } x) = x \rangle$
by (*simp add: pointfree-idE*)

lemma *inv-swap*[simp]: $\langle \text{inv swap} = \text{swap} \rangle$
by (*meson inv-unique-comp swap-o-swap*)

lemma *register-pair-Fst*:
assumes $\langle \text{compatible } F \ G \rangle$
shows $\langle (F;G) \circ Fst = F \rangle$
using *assms* **by** (*auto intro!: ext simp: Fst-def register-pair-apply compatible-register2*)

lemma *register-pair-Snd*:
assumes $\langle \text{compatible } F \ G \rangle$
shows $\langle (F;G) \circ Snd = G \rangle$
using *assms* **by** (*auto intro!: ext simp: Snd-def register-pair-apply compatible-register1*)

lemma *register-Fst-register-Snd*[simp]:
assumes $\langle \text{register } F \rangle$
shows $\langle (F \circ Fst; F \circ Snd) = F \rangle$
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 $\langle \text{register } F \rangle$
shows $\langle (F \circ Snd; F \circ Fst) = F \circ \text{swap} \rangle$
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]: $\langle \text{register } F \rangle \langle \text{register } G \rangle \langle \text{register } H \rangle$
using *assms compatible-def* **by** *auto*
then have [simp]: $\langle \text{preregister } F \rangle \langle \text{preregister } G \rangle \langle \text{preregister } H \rangle$
using *register-preregister* **by** *blast+*
have [simp]: $\langle \text{preregister } (\lambda a. (F;G) \ a \ *_{\mathbf{u}} \ z) \rangle$ **for** z
apply (*rule comp-preregister[unfolded o-def, of $\langle (F;G) \rangle$]*)
by *simp-all*
have [simp]: $\langle \text{preregister } (\lambda a. z \ *_{\mathbf{u}} \ (F;G) \ a) \rangle$ **for** z
apply (*rule comp-preregister[unfolded o-def, of $\langle (F;G) \rangle$]*)
by *simp-all*
have $(F; G) (f \otimes_{\mathbf{u}} g) \ *_{\mathbf{u}} \ H \ h = H \ h \ *_{\mathbf{u}} \ (F; G) (f \otimes_{\mathbf{u}} g)$ **for** $f \ g \ h$
proof –
have $FH: F \ f \ *_{\mathbf{u}} \ H \ h = H \ h \ *_{\mathbf{u}} \ F \ f$
using *assms compatible-def* **by** *metis*
have $GH: G \ g \ *_{\mathbf{u}} \ H \ h = H \ h \ *_{\mathbf{u}} \ G \ g$
using *assms compatible-def* **by** *metis*
have $\langle (F; G) (f \otimes_{\mathbf{u}} g) \ *_{\mathbf{u}} \ (H \ h) = F \ f \ *_{\mathbf{u}} \ G \ g \ *_{\mathbf{u}} \ H \ h \rangle$
using $\langle \text{compatible } F \ G \rangle$ **by** (*subst register-pair-apply, auto*)
also have $\langle \dots = H \ h \ *_{\mathbf{u}} \ F \ f \ *_{\mathbf{u}} \ G \ g \rangle$
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) o swap)⟩
    by simp
  show ⟨preregister (B; A)⟩
    by (metis (no-types, lifting) assms compatible-sym register-preregister pair-is-register)
  show ⟨((A; B) o 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  $\Phi$  where  $\langle \Phi \ c \ d = ((F;F') \otimes_r (G;G')) \circ \text{reorder} \circ (\lambda \sigma. \sigma \otimes_u (c \otimes_u d)) \rangle$  for  $c \ d$ 
have [simp]:  $\langle \text{preregister} (\Phi \ c \ d) \rangle$  for  $c \ d$ 
  unfolding  $\Phi$ -def
  by (auto intro: register-preregister)
have  $\langle \Phi \ c \ d (a \otimes_u b) = (F \otimes_r G) (a \otimes_u b) *_u (F' \otimes_r G') (c \otimes_u d) \rangle$  for  $a \ b \ c \ d$ 
  unfolding  $\Phi$ -def by (auto simp: register-pair-apply tensor-update-mult)
then have  $\Phi 1$ :  $\langle \Phi \ c \ d \ \sigma = (F \otimes_r G) \ \sigma *_u (F' \otimes_r G') (c \otimes_u d) \rangle$  for  $c \ d \ \sigma$ 
  apply (rule-tac fun-cong[of - -  $\sigma$ ])
  apply (rule tensor-extensionality)
  by auto
have  $\langle \Phi \ c \ d (a \otimes_u b) = (F' \otimes_r G') (c \otimes_u d) *_u (F \otimes_r G) (a \otimes_u b) \rangle$  for  $a \ b \ c \ d$ 
  unfolding  $\Phi$ -def apply (auto simp: register-pair-apply)
  by (metis assms(1) assms(2) compatible-def tensor-update-mult)
then have  $\Phi 2$ :  $\langle \Phi \ c \ d \ \sigma = (F' \otimes_r G') (c \otimes_u d) *_u (F \otimes_r G) \ \sigma \rangle$  for  $c \ d \ \sigma$ 
  apply (rule-tac fun-cong[of - -  $\sigma$ ])
  apply (rule tensor-extensionality)
  by auto
from  $\Phi 1 \ \Phi 2$  have  $\langle (F \otimes_r G) \ \sigma *_u (F' \otimes_r G') \ \tau = (F' \otimes_r G') \ \tau *_u (F \otimes_r G) \ \sigma \rangle$  for  $\tau \ \sigma$ 
  apply (rule-tac fun-cong[of - -  $\tau$ ])
  apply (rule tensor-extensionality)
  by auto
then show ?thesis
  apply (rule compatibleI[rotated -I])
  by auto
qed

```

2.8 Associativity of the tensor product

definition $\text{assoc} :: \langle ('a::\text{domain} \times 'b::\text{domain}) \times 'c::\text{domain} \rangle \text{update} \Rightarrow \langle 'a \times ('b \times 'c) \rangle \text{update}$ **where**
 $\langle \text{assoc} = ((Fst; Snd \circ Fst); Snd \circ Snd) \rangle$

lemma assoc-is-hom [simp]: $\langle \text{preregister } \text{assoc} \rangle$
by (auto simp: assoc-def)

lemma assoc-apply [simp]: $\langle \text{assoc} ((a \otimes_u b) \otimes_u c) = (a \otimes_u (b \otimes_u c)) \rangle$
by (auto simp: assoc-def register-pair-apply Fst-def Snd-def tensor-update-mult)

definition $\text{assoc}' :: \langle 'a \times ('b \times 'c) \rangle \text{update} \Rightarrow \langle ('a::\text{domain} \times 'b::\text{domain}) \times 'c::\text{domain} \rangle \text{update}$ **where**
 $\langle \text{assoc}' = (Fst \circ Fst; (Fst \circ Snd; Snd)) \rangle$

lemma $\text{assoc}'\text{-is-hom}$ [simp]: $\langle \text{preregister } \text{assoc}' \rangle$
by (auto simp: assoc'-def)

lemma $\text{assoc}'\text{-apply}$ [simp]: $\langle \text{assoc}' (a \otimes_u (b \otimes_u c)) = ((a \otimes_u b) \otimes_u c) \rangle$
by (auto simp: assoc'-def register-pair-apply Fst-def Snd-def tensor-update-mult)

lemma register-assoc [simp]: $\langle \text{register } \text{assoc} \rangle$
unfolding assoc -def
by force

lemma $\text{register-assoc}'$ [simp]: $\langle \text{register } \text{assoc}' \rangle$
unfolding assoc' -def
by force

lemma pair-o-assoc [simp]:
assumes [simp]: $\langle \text{compatible } F \ G \rangle \langle \text{compatible } G \ H \rangle \langle \text{compatible } F \ H \rangle$
shows $\langle (F; (G; H)) \circ \text{assoc} = ((F; G); H) \rangle$
proof (rule tensor-extensionality3')
show $\langle \text{register} ((F; (G; H)) \circ \text{assoc}) \rangle$
by simp
show $\langle \text{register} ((F; G); H) \rangle$

by *simp*
 show $\langle ((F; (G; H)) \circ \text{assoc}) ((f \otimes_u g) \otimes_u h) = ((F; G); H) ((f \otimes_u g) \otimes_u h) \rangle$ for $f g h$
 by (*simp add: register-pair-apply assoc-apply comp-update-assoc*)
 qed

lemma *pair-o-assoc'*[*simp*]:
 assumes [*simp*]: $\langle \text{compatible } F G \rangle \langle \text{compatible } G H \rangle \langle \text{compatible } F H \rangle$
 shows $\langle ((F; G); H) \circ \text{assoc}' = (F; (G; H)) \rangle$
proof (*rule tensor-extensionality3*)
 show $\langle \text{register } (((F; G); H) \circ \text{assoc}') \rangle$
 by *simp*
 show $\langle \text{register } (F; (G; H)) \rangle$
 by *simp*
 show $\langle (((F; G); H) \circ \text{assoc}') (f \otimes_u g \otimes_u h) = (F; (G; H)) (f \otimes_u g \otimes_u h) \rangle$ for $f g h$
 by (*simp add: register-pair-apply assoc'-apply comp-update-assoc*)
 qed

lemma *assoc'-o-assoc*[*simp*]: $\langle \text{assoc}' \circ \text{assoc} = \text{id} \rangle$
apply (*rule tensor-extensionality3'*)
 by *auto*

lemma *assoc'-assoc*[*simp*]: $\langle \text{assoc}' (\text{assoc } x) = x \rangle$
 by (*simp add: pointfree-idE*)

lemma *assoc-o-assoc'*[*simp*]: $\langle \text{assoc} \circ \text{assoc}' = \text{id} \rangle$
apply (*rule tensor-extensionality3*)
 by *auto*

lemma *assoc-assoc'*[*simp*]: $\langle \text{assoc} (\text{assoc}' x) = x \rangle$
 by (*simp add: pointfree-idE*)

lemma *inv-assoc*[*simp*]: $\langle \text{inv } \text{assoc} = \text{assoc}' \rangle$
using *assoc'-o-assoc assoc-o-assoc' inv-unique-comp* by *blast*

lemma *inv-assoc'*[*simp*]: $\langle \text{inv } \text{assoc}' = \text{assoc} \rangle$
 by (*simp add: inv-equality*)

lemma [*simp*]: $\langle \text{bij } \text{assoc} \rangle$
using *assoc'-o-assoc assoc-o-assoc' o-bij* by *blast*

lemma [*simp*]: $\langle \text{bij } \text{assoc}' \rangle$
using *assoc'-o-assoc assoc-o-assoc' o-bij* by *blast*

2.9 Iso-registers

definition $\langle \text{iso-register } F \longleftrightarrow \text{register } F \wedge (\exists G. \text{register } G \wedge F \circ G = \text{id} \wedge G \circ F = \text{id}) \rangle$
 for $F :: \langle \cdot \rangle :: \text{domain update} \Rightarrow \cdot :: \text{domain update} \rangle$

lemma *iso-registerI*:
 assumes $\langle \text{register } F \rangle \langle \text{register } G \rangle \langle F \circ G = \text{id} \rangle \langle G \circ F = \text{id} \rangle$
 shows $\langle \text{iso-register } F \rangle$
using *assms(1) assms(2) assms(3) assms(4) iso-register-def* by *blast*

lemma *iso-register-inv*: $\langle \text{iso-register } F \Longrightarrow \text{iso-register } (\text{inv } F) \rangle$
 by (*metis inv-unique-comp iso-register-def*)

lemma *iso-register-inv-comp1*: $\langle \text{iso-register } F \Longrightarrow \text{inv } F \circ F = \text{id} \rangle$
using *inv-unique-comp iso-register-def* by *blast*

lemma *iso-register-inv-comp2*: $\langle \text{iso-register } F \Longrightarrow F \circ \text{inv } F = \text{id} \rangle$
using *inv-unique-comp iso-register-def* by *blast*

lemma *iso-register-id*[simp]: $\langle \text{iso-register } id \rangle$
by (*simp add: iso-register-def*)

lemma *iso-register-is-register*: $\langle \text{iso-register } F \implies \text{register } F \rangle$
using *iso-register-def* **by** *blast*

lemma *iso-register-comp*[simp]:
assumes [simp]: $\langle \text{iso-register } F \rangle \langle \text{iso-register } G \rangle$
shows $\langle \text{iso-register } (F \circ G) \rangle$

proof –

from *assms* **obtain** $F' G'$ **where** [simp]: $\langle \text{register } F' \rangle \langle \text{register } G' \rangle \langle F \circ F' = id \rangle \langle F' \circ F = id \rangle$
 $\langle G \circ G' = id \rangle \langle G' \circ G = id \rangle$

by (*meson iso-register-def*)

show *?thesis*

apply (*rule iso-registerI*[**where** $G = \langle G' \circ F' \rangle$])

apply (*auto simp: register-tensor-is-register iso-register-is-register register-tensor-distrib*)

apply (*metis* $\langle F \circ F' = id \rangle \langle G \circ G' = id \rangle$ *fcomp-assoc fcomp-comp id-fcomp*)

by (*metis* (*no-types, lifting*) $\langle F \circ F' = id \rangle \langle F' \circ F = id \rangle \langle G' \circ G = id \rangle$ *fun.map-comp inj-iff inv-unique-comp o-inv-o-cancel*)

qed

lemma *iso-register-tensor-is-iso-register*[simp]:
assumes [simp]: $\langle \text{iso-register } F \rangle \langle \text{iso-register } G \rangle$
shows $\langle \text{iso-register } (F \otimes_r G) \rangle$

proof –

from *assms* **obtain** $F' G'$ **where** [simp]: $\langle \text{register } F' \rangle \langle \text{register } G' \rangle \langle F \circ F' = id \rangle \langle F' \circ F = id \rangle$
 $\langle G \circ G' = id \rangle \langle G' \circ G = id \rangle$

by (*meson iso-register-def*)

show *?thesis*

apply (*rule iso-registerI*[**where** $G = \langle F' \otimes_r G' \rangle$])

by (*auto simp: register-tensor-is-register iso-register-is-register register-tensor-distrib*)

qed

lemma *iso-register-bij*: $\langle \text{iso-register } F \implies \text{bij } F \rangle$
using *iso-register-def o-bij* **by** *auto*

lemma *inv-register-tensor*[simp]:
assumes [simp]: $\langle \text{iso-register } F \rangle \langle \text{iso-register } G \rangle$
shows $\langle \text{inv } (F \otimes_r G) = \text{inv } F \otimes_r \text{inv } G \rangle$
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]: $\langle \text{iso-register } \text{swap} \rangle$
apply (*rule iso-registerI*[*of - swap*])
by *auto*

lemma *iso-register-assoc*[simp]: $\langle \text{iso-register } \text{assoc} \rangle$
apply (*rule iso-registerI*[*of - assoc*'])
by *auto*

lemma *iso-register-assoc'*[simp]: $\langle \text{iso-register } \text{assoc}' \rangle$
apply (*rule iso-registerI*[*of - assoc*])
by *auto*

definition $\langle \text{equivalent-registers } F G \iff (\text{register } F \wedge (\exists I. \text{iso-register } I \wedge F \circ I = G)) \rangle$
for $F G :: \langle -:: \text{domain update} \Rightarrow -:: \text{domain update} \rangle$

lemma *iso-register-equivalent-id*[simp]: $\langle \text{equivalent-registers } id F \iff \text{iso-register } F \rangle$
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 o I = G⟩ ⟨G o 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) o (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) o (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 ^
            \n ^ 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 ^
            \n ^ 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 it *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*: $\langle \text{compatible } F \ G \implies \text{iso-register } (F;G) \implies \text{compatible } F \ H \implies \text{iso-register } (F;H) \implies \text{equivalent-registers } G \ H \rangle$
for $F :: \langle 'a::\text{domain update} \Rightarrow 'b::\text{domain update} \rangle$ **and** $G :: \langle 'g::\text{domain update} \Rightarrow 'b \ \text{update} \rangle$ **and** $H :: \langle 'h::\text{domain update} \Rightarrow 'b \ \text{update} \rangle$

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 $\langle \text{complements } F \ G \longleftrightarrow \text{compatible } F \ G \wedge \text{iso-register } (F;G) \rangle$

lemma *complementsI*: $\langle \text{compatible } F \ G \implies \text{iso-register } (F;G) \implies \text{complements } F \ G \rangle$
using *complements-def* **by** *blast*

lemma *complements-sym*: $\langle \text{complements } G \ F \rangle$ **if** $\langle \text{complements } F \ G \rangle$

proof (*rule complementsI*)

show [*simp*]: $\langle \text{compatible } G \ F \rangle$

using *compatible-sym complements-def that* **by** *blast*

from that have $\langle \text{iso-register } (F;G) \rangle$

by (*meson complements-def*)

then obtain I **where** [*simp*]: $\langle \text{register } I \rangle$ **and** $\langle (F;G) \circ I = \text{id} \rangle$ **and** $\langle I \circ (F;G) = \text{id} \rangle$

using *iso-register-def* **by** *blast*

have $\langle \text{register } (\text{swap } \circ I) \rangle$

using $\langle \text{register } I \rangle$ *register-comp register-swap* **by** *blast*

moreover have $\langle (G;F) \circ (\text{swap } \circ I) = \text{id} \rangle$

by (*simp add: (F;G) \circ I = id rewriteL-comp-comp*)

moreover have $\langle (\text{swap } \circ I) \circ (G;F) = \text{id} \rangle$

by (*metis (no-types, opaque-lifting) swap-swap (I \circ (F;G) = id) calculation(2) comp-def eq-id-iff*)

ultimately show $\langle \text{iso-register } (G;F) \rangle$

using $\langle \text{compatible } G \ F \rangle$ *iso-register-def pair-is-register* **by** *blast*

qed

definition *complement* :: $\langle ('a::\text{domain update} \Rightarrow 'b::\text{domain update}) \Rightarrow (('a,'b) \ \text{complement-domain update} \Rightarrow 'b \ \text{update}) \rangle$ **where**

$\langle \text{complement } F = (\text{SOME } G :: ('a, 'b) \ \text{complement-domain update} \Rightarrow 'b \ \text{update}. \ \text{compatible } F \ G \wedge \text{iso-register } (F;G)) \rangle$

lemma *register-complement*[*simp*]: $\langle \text{register } (\text{complement } F) \rangle$ **if** $\langle \text{register } F \rangle$

using *complement-exists[OF that]*

by (*metis (no-types, lifting) compatible-def complement-def some-eq-imp*)

lemma *complement-is-complement*:

assumes $\langle \text{register } F \rangle$

shows $\langle \text{complements } F \ (\text{complement } F) \rangle$

using *complement-exists[OF assms]* **unfolding** *complements-def*

by (*metis (mono-tags, lifting) complement-def some-eq-imp*)

lemma *complement-unique*:

assumes $\langle \text{complements } F \ G \rangle$

shows $\langle \text{equivalent-registers } G \ (\text{complement } F) \rangle$

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*]: $\langle \text{register } F \implies \text{compatible } F \ (\text{complement } F) \rangle$

using *complement-is-complement complements-def* by *blast*

lemma *complements-register-tensor*:

assumes [*simp*]: $\langle \text{register } F \rangle \langle \text{register } G \rangle$

shows $\langle \text{complements } (F \otimes_r G) \text{ (complement } F \otimes_r \text{ complement } G) \rangle$

proof (rule *complementsI*)

have *sep4*: $\langle \text{separating } \text{TYPE}('z::\text{domain}) \{(a \otimes_u b) \otimes_u (c \otimes_u d) \mid a \ b \ c \ d. \text{True}\} \rangle$

apply (rule *separating-tensor'*[**where** $A = \{(a \otimes_u b) \mid a \ b. \text{True}\}$ and $B = \{(c \otimes_u d) \mid c \ d. \text{True}\}$])

apply (rule *separating-tensor'*[**where** $A = \text{UNIV}$ and $B = \text{UNIV}$]) **apply** *auto*[3]

apply (rule *separating-tensor'*[**where** $A = \text{UNIV}$ and $B = \text{UNIV}$]) **apply** *auto*[3]

by *auto*

show *compat*: $\langle \text{compatible } (F \otimes_r G) \text{ (complement } F \otimes_r \text{ complement } G) \rangle$

by (*metis* *assms(1)* *assms(2)* *compatible-register-tensor* *complement-is-complement* *complements-def*)

let *?reorder* = $\langle ((Fst \ o \ Fst; \ Snd \ o \ Fst); (Fst \ o \ Snd; \ Snd \ o \ Snd)) \rangle$

have [*simp*]: $\langle \text{register } ?reorder \rangle$

by *auto*

have [*simp*]: $\langle ?reorder \ ((a \otimes_u b) \otimes_u (c \otimes_u d)) = ((a \otimes_u c) \otimes_u (b \otimes_u d)) \rangle$

for $a::\langle 't::\text{domain update} \rangle$ **and** $b::\langle 'u::\text{domain update} \rangle$ **and** $c::\langle 'v::\text{domain update} \rangle$ **and** $d::\langle 'w::\text{domain update} \rangle$

by (*simp* *add*: *register-pair-apply* *Fst-def* *Snd-def* *tensor-update-mult*)

have [*simp*]: $\langle \text{iso-register } ?reorder \rangle$

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 $\langle (F \otimes_r G; \text{complement } F \otimes_r \text{ complement } G) = ((F; \text{complement } F) \otimes_r (G; \text{complement } G)) \ o \ ?reorder \rangle$

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** $\langle \text{iso-register } \dots \rangle$

apply (*auto* *intro!*: *iso-register-comp* *iso-register-tensor-is-iso-register*)

using *assms* *complement-is-complement* *complements-def* **by** *blast+*

ultimately **show** $\langle \text{iso-register } (F \otimes_r G; \text{complement } F \otimes_r \text{ complement } G) \rangle$

by *simp*

qed

definition *is-unit-register* **where**

$\langle \text{is-unit-register } U \iff \text{complements } U \ \text{id} \rangle$

lemma *register-unit-register*[*simp*]: $\langle \text{is-unit-register } U \implies \text{register } U \rangle$

by (*simp* *add*: *compatible-def* *complements-def* *is-unit-register-def*)

lemma *unit-register-compatible*[*simp*]: $\langle \text{compatible } U \ X \rangle$ **if** $\langle \text{is-unit-register } U \rangle \langle \text{register } X \rangle$

by (*metis* *compatible-comp-right* *complements-def* *id-comp* *is-unit-register-def* *that(1)* *that(2)*)

lemma *unit-register-compatible'*[*simp*]: $\langle \text{compatible } X \ U \rangle$ **if** $\langle \text{is-unit-register } U \rangle \langle \text{register } X \rangle$

using *compatible-sym* *that(1)* *that(2)* *unit-register-compatible* **by** *blast*

lemma *compatible-complement-left*[*simp*]: $\langle \text{register } X \implies \text{compatible } (\text{complement } X) \ X \rangle$

using *compatible-sym* *complement-is-complement* *complements-def* **by** *blast*

lemma *compatible-complement-right*[*simp*]: $\langle \text{register } X \implies \text{compatible } X \ (\text{complement } X) \rangle$

using *complement-is-complement* *complements-def* **by** *blast*

lemma *unit-register-pair*[*simp*]: $\langle \text{equivalent-registers } X \ (U; \ X) \rangle$ **if** [*simp*]: $\langle \text{is-unit-register } U \rangle \langle \text{register } X \rangle$

proof –

have $\langle \text{equivalent-registers } \text{id} \ (U; \ \text{id}) \rangle$

using *complements-def* *is-unit-register-def* *iso-register-equivalent-id* *that(1)* **by** *blast*

also **have** $\langle \text{equivalent-registers } \dots \ (U; \ (X; \ \text{complement } X)) \rangle$

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** $\langle \text{equivalent-registers } \dots \ ((U; \ X); \ \text{complement } X) \rangle$

apply (rule *equivalent-registers-assoc*)

by *auto*

finally have $\langle \text{complements } (U; X) \text{ (complement } X) \rangle$
by (*auto simp: equivalent-registers-def complements-def*)
moreover have $\langle \text{equivalent-registers } (X; \text{complement } X) \text{ id} \rangle$
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*]: $\langle \text{is-unit-register } U \rangle$
assumes [*simp*]: $\langle \text{register } A \rangle$
shows $\langle \text{is-unit-register } (A \circ U) \rangle$
proof –
have $\langle \text{compatible } (A \circ U) (A; \text{complement } A) \rangle$
apply (*auto intro!: compatible3'*)
by (*metis assms(1) assms(2) comp-id compatible-comp-inner complements-def is-unit-register-def*)
then have $\text{compat}[simp]: \langle \text{compatible } (A \circ U) \text{ id} \rangle$
by (*metis assms(2) compatible-comp-right complement-is-complement complements-def iso-register-def*)
have $\langle \text{equivalent-registers } (A \circ U; \text{id}) (A \circ U; (A; \text{complement } A)) \rangle$
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 $\langle \text{equivalent-registers } \dots ((A \circ U; A \circ \text{id}); \text{complement } A) \rangle$
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 $\langle \text{equivalent-registers } \dots (A \circ (U; \text{id}); \text{complement } A) \rangle$
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 $\langle \text{equivalent-registers } \dots (A \circ \text{id}; \text{complement } A) \rangle$
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 $\langle \text{equivalent-registers } \dots \text{id} \rangle$
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*]: $\langle \text{is-unit-register } U \rangle$
assumes [*simp*]: $\langle \text{iso-register } A \rangle$
shows $\langle \text{is-unit-register } (U \circ A) \rangle$
proof (*unfold is-unit-register-def, rule complementsI*)
show $\langle \text{compatible } (U \circ A) \text{ id} \rangle$
by (*simp add: iso-register-is-register*)
have *1*: $\langle \text{iso-register } ((U; \text{id}) \circ A \otimes_r \text{id}) \rangle$
by (*meson assms(1) assms(2) complements-def is-unit-register-def iso-register-comp iso-register-id iso-register-tensor-is-iso-register*)
have *2*: $\langle \text{id} \circ ((U; \text{id}) \circ A \otimes_r \text{id}) = (U \circ A; \text{id}) \rangle$
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 $\langle \text{iso-register } (U \circ A; \text{id}) \rangle$
using *1 2* **by auto**
qed

lemma *unit-register-unique:*

assumes $\langle \text{is-unit-register } F \rangle$
assumes $\langle \text{is-unit-register } G \rangle$
shows $\langle \text{equivalent-registers } F G \rangle$
proof –
have $\langle \text{complements } F \text{ id} \rangle \langle \text{complements } G \text{ id} \rangle$

```

  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 ((λd. tensor-update id-update d) o G)⟩
  by (simp add: assms(2) unit-register-compose-left)
  moreover have ⟨is-unit-register ((λc. tensor-update c id-update) o F)⟩
  using assms(1) register-tensor-left unit-register-compose-left by blast
  ultimately have ⟨equivalent-registers ((λd. tensor-update id-update d) o G) ((λc. 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 $\langle \text{compatible } F \ G \rangle$

shows $\langle \text{compatible } F \ (\text{complement } (F;G)) \rangle$

by (*metis* *assms* *compatible-comp-left* *compatible-complement-right* *pair-is-register* *register-Fst* *register-pair-Fst*)

lemma *compatible-complement-pair2*:

assumes [*simp*]: $\langle \text{compatible } F \ G \rangle$

shows $\langle \text{compatible } G \ (\text{complement } (F;G)) \rangle$

proof –

have $\langle \text{compatible } (F;G) \ (\text{complement } (F;G)) \rangle$

by *simp*

then have $\langle \text{compatible } ((F;G) \ o \ \text{Snd}) \ (\text{complement } (F;G)) \rangle$

by *auto*

then show *?thesis*

by (*auto* *simp*: *register-pair-Snd*)

qed

lemma *equivalent-complements*:

assumes $\langle \text{complements } F \ G \rangle$

assumes $\langle \text{equivalent-registers } G \ G' \rangle$

shows $\langle \text{complements } F \ G' \rangle$

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*]: $\langle \text{compatible } F \ G \rangle$

shows $\langle \text{complements } F \ (G; \text{complement } (F;G)) \rangle$

proof (*rule* *complementsI*)

have $\langle \text{equivalent-registers } (F; (G; \text{complement } (F;G))) \ ((F;G); \text{complement } (F;G)) \rangle$

apply (*rule* *equivalent-registers-assoc*)

by (*auto* *simp* *add*: *compatible-complement-pair1* *compatible-complement-pair2*)

also have $\langle \text{equivalent-registers } \dots \ \text{id} \rangle$

by (*meson* *assms* *complement-is-complement* *complements-def* *equivalent-registers-sym* *iso-register-equivalent-id* *pair-is-register*)

finally show $\langle \text{iso-register } (F;(G;\text{complement } (F;G))) \rangle$

using *equivalent-registers-sym* *iso-register-equivalent-id* **by** *blast*

show $\langle \text{compatible } F \ (G;\text{complement } (F;G)) \rangle$

using *assms* *compatible3'* *compatible-complement-pair1* *compatible-complement-pair2* **by** *blast*

qed

lemma *equivalent-registers-complement*:

assumes $\langle \text{equivalent-registers } F \ G \rangle$

shows $\langle \text{equivalent-registers } (\text{complement } F) \ (\text{complement } G) \rangle$

proof –

have $\langle \text{complements } F \ (\text{complement } F) \rangle$

using *assms* *complement-is-complement* *equivalent-registers-register-left* **by** *blast*

with *assms* **have** $\langle \text{complements } G \ (\text{complement } F) \rangle$

by (*meson* *complements-sym* *equivalent-complements*)

then show *?thesis*

by (*rule* *complement-unique*)

qed

lemma *complements-complement-pair'*:

assumes [*simp*]: $\langle \text{compatible } F \ G \rangle$

shows $\langle \text{complements } G \ (F; \text{complement } (F;G)) \rangle$

proof –

have $\langle \text{equivalent-registers } (F;G) \ (G;F) \rangle$

apply (*rule* *equivalent-registersI*[**where** $I = \text{swap}$])

by *auto*

then have $\langle \text{equivalent-registers } (\text{complement } (F;G)) (\text{complement } (G;F)) \rangle$
by (rule *equivalent-registers-complement*)
then have $\langle \text{equivalent-registers } (F; (\text{complement } (F;G))) (F; (\text{complement } (G;F))) \rangle$
apply (rule *equivalent-registers-pair-right*[rotated])
using *assms compatible-complement-pair1* **by** *blast*
moreover have $\langle \text{complements } G (F; \text{complement } (G;F)) \rangle$
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*]: $\langle \text{register } F \rangle \langle \text{register } G \rangle$
shows $\langle \text{complements } (F \circ G) (\text{complement } F; F \circ \text{complement } G) \rangle$
proof (rule *complementsI*)
show $\langle \text{compatible } (F \circ G) (\text{complement } F; F \circ \text{complement } G) \rangle$
by *auto*
have $\langle \text{equivalent-registers } (F \circ G; (\text{complement } F; F \circ \text{complement } G)) (F \circ G; (F \circ \text{complement } G; \text{complement } F)) \rangle$
apply (rule *equivalent-registersI*[**where** $I = \langle \text{id} \otimes_r \text{swap} \rangle$])
by (*auto intro!*: *iso-register-tensor-is-iso-register simp: pair-o-tensor*)
also have $\langle \text{equivalent-registers } \dots ((F \circ G; F \circ \text{complement } G); \text{complement } F) \rangle$
apply (rule *equivalent-registersI*[**where** $I = \text{assoc}$])
by (*auto intro!*: *iso-register-tensor-is-iso-register simp: pair-o-tensor*)
also have $\langle \text{equivalent-registers } \dots (F \circ (G; \text{complement } G); \text{complement } F) \rangle$
by (*metis* (*no-types, lifting*) *assms(1) assms(2) calculation compatible-complement-right equivalent-registers-sym equivalent-registers-trans register-comp-pair*)
also have $\langle \text{equivalent-registers } \dots (F \circ \text{id}; \text{complement } F) \rangle$
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 $\langle \text{equivalent-registers } \dots \text{id} \rangle$
by (*metis* *assms(1) comp-id complement-is-complement complements-def equivalent-registers-def iso-register-def*)
finally show $\langle \text{iso-register } (F \circ G; (\text{complement } F; F \circ \text{complement } G)) \rangle$
using *equivalent-registers-sym iso-register-equivalent-id* **by** *blast*
qed

lemma *complements-Fst-Snd*[*simp*]: $\langle \text{complements } \text{Fst } \text{Snd} \rangle$
by (*auto intro!*: *complementsI simp: pair-Fst-Snd*)

lemma *complements-Snd-Fst*[*simp*]: $\langle \text{complements } \text{Snd } \text{Fst} \rangle$
by (*auto intro!*: *complementsI simp flip: swap-def*)

lemma *compatible-unit-register*[*simp*]: $\langle \text{register } F \implies \text{compatible } F \text{ unit-register} \rangle$
using *compatible-sym unit-register-compatible unit-register-is-unit-register* **by** *blast*

lemma *complements-id-unit-register*[*simp*]: $\langle \text{complements } \text{id} \text{ unit-register} \rangle$
using *complements-sym is-unit-register-def unit-register-is-unit-register* **by** *blast*

lemma *complements-iso-unit-register*: $\langle \text{iso-register } I \implies \text{is-unit-register } U \implies \text{complements } I \ U \rangle$
using *complements-sym equivalent-complements is-unit-register-def iso-register-equivalent-id* **by** *blast*

lemma *iso-register-complement-is-unit-register*[*simp*]:
assumes $\langle \text{iso-register } F \rangle$
shows $\langle \text{is-unit-register } (\text{complement } F) \rangle$
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 $\langle \text{is-unit-register } - \rangle$]

no-notation *comp-update* (**infixl** *_u 55)
no-notation *tensor-update* (**infixr** ⊗_u 70)

end

5 Classical instantiation of registers

theory *Axioms-Classical*

imports *Main*

begin

type-synonym *'a update* = $\langle 'a \rightarrow 'a \rangle$

lemma *id-update-left*: $\text{Some } \circ_m a = a$

by (*auto intro!*: *ext simp add: map-comp-def option.case-eq-if*)

lemma *id-update-right*: $a \circ_m \text{Some} = a$

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* = $\langle 'a \text{ update} \Rightarrow 'b \text{ update} \rangle$

definition *preregister* :: $\langle ('a, 'b) \text{ preregister} \Rightarrow \text{bool} \rangle$ **where**

$\langle \text{preregister } F \longleftrightarrow (\exists g s. \forall a m. F a m = (\text{case } a (g m) \text{ of } \text{None} \Rightarrow \text{None} \mid \text{Some } x \Rightarrow s x m)) \rangle$

lemma *id-preregister*: $\langle \text{preregister } \text{id} \rangle$

unfolding *preregister-def*

apply (*rule exI[of - $\langle \lambda m. m \rangle$]*)

apply (*rule exI[of - $\langle \lambda a m. \text{Some } a \rangle$]*)

by (*simp add: option.case-eq-if*)

lemma *preregister-mult-right*: $\langle \text{preregister } (\lambda a. a \circ_m z) \rangle$

unfolding *preregister-def*

apply (*rule exI[of - $\langle \lambda m. \text{the } (z m) \rangle$]*)

apply (*rule exI[of - $\langle \lambda x m. \text{case } z m \text{ of } \text{None} \Rightarrow \text{None} \mid - \Rightarrow \text{Some } x \rangle$]*)

by (*auto simp add: option.case-eq-if*)

lemma *preregister-mult-left*: $\langle \text{preregister } (\lambda a. z \circ_m a) \rangle$

unfolding *preregister-def*

apply (*rule exI[of - $\langle \lambda m. m \rangle$]*)

apply (*rule exI[of - $\langle \lambda x m. z x \rangle$]*)

by (*auto simp add: option.case-eq-if*)

lemma *comp-preregister*: *preregister* ($G \circ F$) **if** *preregister* *F* **and** $\langle \text{preregister } G \rangle$

proof –

from $\langle \text{preregister } F \rangle$

obtain *sF gF* **where** $F: \langle F a m = (\text{case } a (gF m) \text{ of } \text{None} \Rightarrow \text{None} \mid \text{Some } x \Rightarrow sF x m) \rangle$ **for** *a m*

using *preregister-def* **by** *blast*

from $\langle \text{preregister } G \rangle$

obtain *sG gG* **where** $G: \langle G a m = (\text{case } a (gG m) \text{ of } \text{None} \Rightarrow \text{None} \mid \text{Some } x \Rightarrow sG x m) \rangle$ **for** *a m*

using *preregister-def* **by** *blast*

define *s g* **where** $\langle s a m = (\text{case } sF a (gG m) \text{ of } \text{None} \Rightarrow \text{None} \mid \text{Some } x \Rightarrow sG x m) \rangle$

and $\langle g m = gF (gG m) \rangle$ **for** *a m*

have $\langle (G \circ F) a m = (\text{case } a (g m) \text{ of } \text{None} \Rightarrow \text{None} \mid \text{Some } x \Rightarrow s x m) \rangle$ **for** *a m*

unfolding *F G s-def g-def*

by (*auto simp add: option.case-eq-if*)

then show *preregister* ($G \circ F$)

using *preregister-def* **by** *blast*

qed

definition *tensor-update* :: $\langle 'a \text{ update} \Rightarrow 'b \text{ update} \Rightarrow ('a \times 'b) \text{ update} \rangle$ **where**

$\langle \text{tensor-update } a b m = (\text{case } a (fst m) \text{ of } \text{None} \Rightarrow \text{None} \mid \text{Some } x \Rightarrow (\text{case } b (snd m) \text{ of } \text{None} \Rightarrow \text{None} \mid \text{Some } y \Rightarrow \text{Some } (x, y))) \rangle$

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

definition *update1* :: $\langle 'a \Rightarrow 'a \Rightarrow 'a \text{ update} \rangle$ **where**
 $\langle \text{update1 } x \text{ } y \text{ } m = (\text{if } m=x \text{ then } \text{Some } y \text{ else } \text{None}) \rangle$

lemma *update1-extensionality*:

assumes $\langle \text{preregister } F \rangle$
assumes $\langle \text{preregister } G \rangle$
assumes $\langle FGeq: \langle \bigwedge x \text{ } y. F (\text{update1 } x \text{ } y) = G (\text{update1 } x \text{ } y) \rangle \rangle$
shows $F = G$

proof (*rule ccontr*)

assume $\text{neq}: \langle F \neq G \rangle$

then obtain $z \text{ } m$ **where** $\text{neq}': \langle F \text{ } z \text{ } m \neq G \text{ } z \text{ } m \rangle$

apply *atomize-elim* **by** *auto*

obtain $gF \text{ } sF$ **where** $gsF: \langle F \text{ } z \text{ } m = (\text{case } z \text{ } (gF \text{ } m) \text{ of } \text{None} \Rightarrow \text{None} \mid \text{Some } x \Rightarrow sF \text{ } x \text{ } m) \rangle$ **for** $z \text{ } m$

using $\langle \text{preregister } F \rangle$ *preregister-def* **by** *blast*

obtain $gG \text{ } sG$ **where** $gsG: \langle G \text{ } z \text{ } m = (\text{case } z \text{ } (gG \text{ } m) \text{ of } \text{None} \Rightarrow \text{None} \mid \text{Some } x \Rightarrow sG \text{ } x \text{ } m) \rangle$ **for** $z \text{ } m$

using $\langle \text{preregister } G \rangle$ *preregister-def* **by** *blast*

consider (*abeq* x **where** $\langle z \text{ } (gF \text{ } m) = \text{Some } x \rangle \langle z \text{ } (gG \text{ } m) = \text{Some } x \rangle \langle gF \text{ } m = gG \text{ } m \rangle$

| (*abnone*) $\langle z \text{ } (gG \text{ } m) = \text{None} \rangle \langle z \text{ } (gF \text{ } m) = \text{None} \rangle$

| (*neqF*) x **where** $\langle gF \text{ } m \neq gG \text{ } m \rangle \langle F \text{ } z \text{ } m = \text{Some } x \rangle$

| (*neqG*) y **where** $\langle gF \text{ } m \neq gG \text{ } m \rangle \langle G \text{ } z \text{ } m = \text{Some } y \rangle$

| (*neqNone*) $\langle gF \text{ } m \neq gG \text{ } m \rangle \langle F \text{ } z \text{ } m = \text{None} \rangle \langle G \text{ } z \text{ } m = \text{None} \rangle$

apply *atomize-elim* **by** (*metis option.exhaust-sel*)

then show *False*

proof *cases*

case (*abeq* x)

then have $\langle F \text{ } z \text{ } m = sF \text{ } x \text{ } m \rangle$ **and** $\langle G \text{ } z \text{ } m = sG \text{ } x \text{ } m \rangle$

by (*simp-all add: gsF gsG*)

moreover have $\langle F (\text{update1 } (gF \text{ } m) \text{ } x) \text{ } m = sF \text{ } x \text{ } m \rangle$

by (*simp add: gsF update1-def*)

moreover have $\langle G (\text{update1 } (gG \text{ } m) \text{ } x) \text{ } m = sG \text{ } x \text{ } m \rangle$

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 $\langle F (\text{update1 } (gF \text{ } m) \text{ } (\text{the } (z \text{ } (gF \text{ } m)))) \text{ } m = F \text{ } z \text{ } m \rangle$

by (*metis gsF neqF(2) option.case-eq-if option.simps(3) option.simps(5) update1-def*)

moreover have $\langle G (\text{update1 } (gF \text{ } m) \text{ } (\text{the } (z \text{ } (gF \text{ } m)))) \text{ } m = \text{None} \rangle$

by (*metis gsG neqF(1) option.case-eq-if update1-def*)

ultimately show *False*

using *FGeq* **by** *force*

next

case *neqG*

moreover

have $\langle G (\text{update1 } (gG \text{ } m) \text{ } (\text{the } (z \text{ } (gG \text{ } m)))) \text{ } m = G \text{ } z \text{ } m \rangle$

by (*metis gsG neqG(2) option.case-eq-if option.distinct(1) option.simps(5) update1-def*)

moreover have $\langle F (\text{update1 } (gG \text{ } m) \text{ } (\text{the } (z \text{ } (gG \text{ } m)))) \text{ } m = \text{None} \rangle$

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 $\langle \text{preregister } F \rangle$
assumes $\langle \text{preregister } G \rangle$
assumes $\langle FGeg: \langle \bigwedge a b. F (\text{tensor-update } a b) = G (\text{tensor-update } a b) \rangle$
shows $F = G$

proof –

have $\langle F (\text{update1 } x y) = G (\text{update1 } x y) \rangle$ for $x y$
using $FGeg[\text{of } \langle \text{update1 } (\text{fst } x) (\text{fst } y) \rangle \langle \text{update1 } (\text{snd } x) (\text{snd } y) \rangle]$
apply $(\text{auto intro! ext simp: tensor-update-def[abs-def] update1-def[abs-def]})$
by $(\text{smt } (z3) \text{ assms}(1) \text{ assms}(2) \text{ option.case}(2) \text{ option.case-eq-if preregister-def prod.collapse})$
with $\text{assms}(1,2)$ show $F = G$
by $(\text{rule update1-extensionality})$

qed

definition *valid-getter-setter* $g s \longleftrightarrow$

$(\forall b. b = s (g b) b) \wedge (\forall a b. g (s a b) = a) \wedge (\forall a a' b. s a (s a' b) = s a b)$

definition $\langle \text{register-from-getter-setter } g s a m = (\text{case } a (g m) \text{ of } \text{None} \Rightarrow \text{None} \mid \text{Some } x \Rightarrow \text{Some } (s x m)) \rangle$

definition $\langle \text{register-apply } F a = \text{the } o F (\text{Some } o a) \rangle$

definition $\langle \text{setter } F a m = \text{register-apply } F (\lambda \cdot. a) m \rangle$ for $F :: \langle 'a \text{ update} \Rightarrow 'b \text{ update} \rangle$

definition $\langle \text{getter } F m = (\text{THE } x. \text{setter } F x m = m) \rangle$ for $F :: \langle 'a \text{ update} \Rightarrow 'b \text{ update} \rangle$

lemma

assumes $\langle \text{valid-getter-setter } g s \rangle$
shows $\text{getter-of-register-from-getter-setter}[\text{simp}]: \langle \text{getter } (\text{register-from-getter-setter } g s) = g \rangle$
and $\text{setter-of-register-from-getter-setter}[\text{simp}]: \langle \text{setter } (\text{register-from-getter-setter } g s) = s \rangle$

proof –

define $g' s'$ where $\langle g' = \text{getter } (\text{register-from-getter-setter } g s) \rangle$
and $\langle s' = \text{setter } (\text{register-from-getter-setter } g s) \rangle$

show $\langle s' = s \rangle$

by $(\text{auto intro! ext simp: } s'\text{-def setter-def register-apply-def register-from-getter-setter-def})$

moreover show $\langle g' = g \rangle$

proof $(\text{rule ext, rename-tac } m)$

fix m

have $\langle g' m = (\text{THE } x. s x m = m) \rangle$

by $(\text{auto intro! ext simp: } g'\text{-def } s'\text{-def[symmetric] } \langle s' = s \rangle \text{ getter-def register-apply-def register-from-getter-setter-def})$

moreover have $\langle s (g m) m = m \rangle$

by $(\text{metis assms valid-getter-setter-def})$

moreover have $\langle x = x' \rangle$ if $\langle s x m = m \rangle \langle s x' m = m \rangle$ for $x x'$

by $(\text{metis assms that}(1) \text{ that}(2) \text{ valid-getter-setter-def})$

ultimately show $\langle g' m = g m \rangle$

by $(\text{simp add: Uniq-def the1-equality'})$

qed

qed

definition *register* $:: \langle ('a, 'b) \text{preregister} \Rightarrow \text{bool} \rangle$ where

$\langle \text{register } F \longleftrightarrow (\exists g s. F = \text{register-from-getter-setter } g s \wedge \text{valid-getter-setter } g s) \rangle$

lemma *register-of-id*: $\langle \text{register } F \Longrightarrow F \text{Some} = \text{Some} \rangle$

by $(\text{auto simp add: register-def valid-getter-setter-def register-from-getter-setter-def})$

lemma *register-id*: $\langle \text{register id} \rangle$

unfolding *register-def*

apply $(\text{rule exI[of - id], rule exI[of - } \langle \lambda a m. a \rangle])$

by $(\text{auto intro! ext simp: option.case-eq-if register-from-getter-setter-def valid-getter-setter-def})$

lemma *register-tensor-left*: $\langle \text{register } (\lambda a. \text{tensor-update } a \text{Some}) \rangle$

apply $(\text{auto simp: register-def})$

apply $(\text{rule exI[of - fst]})$

apply $(\text{rule exI[of - } \langle \lambda x' (x, y). (x', y) \rangle])$

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: $\langle \text{register } (\lambda a. \text{tensor-update } \text{Some } a) \rangle$
 apply (auto simp: register-def)
 apply (rule exI[of - snd])
 apply (rule exI[of - $\langle \lambda y'. (x, y) \rangle$])
 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 $\langle \text{register } F \rangle$

proof –

from $\langle \text{register } F \rangle$
 obtain $s g$ **where** $F: \langle F a m = (\text{case } a (g m) \text{ of } \text{None} \Rightarrow \text{None} \mid \text{Some } x \Rightarrow \text{Some } (s x m)) \rangle$ **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 - $\langle \lambda x m. \text{Some } (s x m) \rangle$])
 using F **by** simp

qed

lemma register-comp: register $(G \circ F)$ if $\langle \text{register } F \rangle$ **and** $\langle \text{register } G \rangle$

for $F :: ('a, 'b)$ preregister **and** $G :: ('b, 'c)$ preregister

proof –

from $\langle \text{register } F \rangle$
 obtain $sF gF$ **where** $F: \langle F a m = (\text{case } a (gF m) \text{ of } \text{None} \Rightarrow \text{None} \mid \text{Some } x \Rightarrow \text{Some } (sF x m)) \rangle$
and $\text{valid}F: \langle \text{valid-getter-setter } gF sF \rangle$ **for** $a m$
 unfolding register-def register-from-getter-setter-def **by** blast
 from $\langle \text{register } G \rangle$
 obtain $sG gG$ **where** $G: \langle G a m = (\text{case } a (gG m) \text{ of } \text{None} \Rightarrow \text{None} \mid \text{Some } x \Rightarrow \text{Some } (sG x m)) \rangle$
and $\text{valid}G: \langle \text{valid-getter-setter } gG sG \rangle$ **for** $a m$
 unfolding register-def register-from-getter-setter-def **by** blast
 define $s g$ **where** $\langle s a m = sG (sF a (gG m)) m \rangle$ **and** $\langle g m = gF (gG m) \rangle$ **for** $a m$
 have $\langle (G \circ F) a m = (\text{case } a (g m) \text{ of } \text{None} \Rightarrow \text{None} \mid \text{Some } x \Rightarrow \text{Some } (s x m)) \rangle$ **for** $a m$
by (auto simp add: option.case-eq-if $F G s$ -def g -def)
 moreover have $\langle \text{valid-getter-setter } g s \rangle$
using $\text{valid}F \text{ valid}G$ **by** (auto simp: valid-getter-setter-def s -def g -def)
 ultimately show register $(G \circ F)$
 unfolding register-def register-from-getter-setter-def **by** blast

qed

lemma register-mult: register $F \implies F a \circ_m F b = F (a \circ_m b)$

by (auto intro!: ext simp: register-def register-from-getter-setter-def[abs-def] valid-getter-setter-def map-comp-def option.case-eq-if)

definition register-pair ::

$\langle ('a \text{ update} \Rightarrow 'c \text{ update}) \Rightarrow ('b \text{ update} \Rightarrow 'c \text{ update}) \Rightarrow (('a \times 'b) \text{ update} \Rightarrow 'c \text{ update}) \rangle$ **where**
 $\langle \text{register-pair } F G =$
 register-from-getter-setter $(\lambda m. (\text{getter } F m, \text{getter } G m)) (\lambda (a, b) m. \text{setter } F a (\text{setter } G b m)) \rangle$

lemma compatible-setter:

assumes [simp]: $\langle \text{register } F \rangle \langle \text{register } G \rangle$
assumes compat: $\langle \bigwedge a b. F a \circ_m G b = G b \circ_m F a \rangle$
shows $\langle \text{setter } F x \circ \text{setter } G y = \text{setter } G y \circ \text{setter } F x \rangle$
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]: $\langle \text{register } F \rangle \langle \text{register } G \rangle$
assumes $\langle \bigwedge a b. F a \circ_m G b = G b \circ_m F a \rangle$
shows $\langle (\text{register-pair } F G) (\text{tensor-update } a b) = F a \circ_m G b \rangle$

proof –

have $\text{valid}F: \langle \text{valid-getter-setter } (\text{getter } F) (\text{setter } F) \rangle$ **and** $\text{valid}G: \langle \text{valid-getter-setter } (\text{getter } G) (\text{setter } G) \rangle$
by (metis assms getter-of-register-from-getter-setter register-def setter-of-register-from-getter-setter)+

```

then have F: ⟨F = register-from-getter-setter (getter F) (setter F)⟩ and G: ⟨G = register-from-getter-setter
(getter G) (setter G)⟩
  by (metis assms getter-of-register-from-getter-setter register-def setter-of-register-from-getter-setter)+
have gFsG: ⟨getter F (setter G y m) = getter F m⟩ for y m
proof –
  have ⟨getter F (setter G y m) = getter F (setter G y (setter F (getter F m) m))⟩
    using validF by (metis valid-getter-setter-def)
  also have ⟨... = getter F (setter F (getter F m) (setter G y m))⟩
    by (metis (mono-tags, lifting) assms(1) assms(2) assms(3) comp-eq-dest-lhs compatible-setter)
  also have ⟨... = getter F m⟩
    by (metis validF valid-getter-setter-def)
  finally show ?thesis by –
qed

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

lemma register-pair-is-register:
  fixes F :: ⟨'a update ⇒ 'c update⟩ and G
  assumes [simp]: ⟨register F⟩ and [simp]: ⟨register G⟩
  assumes compat: ⟨∧ a b. F a ◦m G b = G b ◦m F a⟩
  shows ⟨register (register-pair F G)⟩
proof –
  have validF: ⟨valid-getter-setter (getter F) (setter F)⟩ and validG: ⟨valid-getter-setter (getter G) (setter G)⟩
    by (metis assms getter-of-register-from-getter-setter register-def setter-of-register-from-getter-setter)+
  then have ⟨valid-getter-setter (λm. (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]: $\langle \text{preregister } (\lambda b :: 'b :: \text{type update. tensor-update } a \ b) \rangle$
for $a :: \langle 'a :: \text{type update} \rangle$
proof –
have $\langle \text{preregister } ((\lambda b1 :: ('a \times 'b) \ \text{update. } (a \otimes_u \text{Some}) *_{\text{u}} b1) \ o \ (\lambda b. \ \text{tensor-update } \text{Some } b)) \rangle$
by (*rule comp-preregister; simp*)
then show *?thesis*
by (*simp add: o-def tensor-update-mult*)
qed

lemma *preregister-tensor-right*[simp]: $\langle \text{preregister } (\lambda a :: 'a :: \text{type update. tensor-update } a \ b) \rangle$
for $b :: \langle 'b :: \text{type update} \rangle$
proof –
have $\langle \text{preregister } ((\lambda a1 :: ('a \times 'b) \ \text{update. } (\text{Some } \otimes_u \ b) *_{\text{u}} a1) \ o \ (\lambda a. \ \text{tensor-update } a \ \text{Some})) \rangle$
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 $\langle \text{register } F \rangle$
shows $\langle \text{register } (\lambda a :: 'a :: \text{type update. } \text{Some } \otimes_u \ F \ a) \rangle$
using *assms* **apply** (*rule register-comp[unfolded o-def]*)
by *simp*

lemma *register-tensor-id-update*[simp]:
assumes $\langle \text{register } F \rangle$
shows $\langle \text{register } (\lambda a :: 'a :: \text{type update. } F \ a \ \otimes_u \ \text{Some}) \rangle$
using *assms* **apply** (*rule register-comp[unfolded o-def]*)
by *simp*

6.4 Tensor product of registers

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

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

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

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

lemma *separating-UNIV*[simp]: $\langle \text{separating } \text{TYPE}(-) \ \text{UNIV} \rangle$
unfolding *separating-def* **by** *auto*

lemma *separating-mono*: $\langle A \subseteq B \Longrightarrow \text{separating } \text{TYPE}('a :: \text{type}) \ A \Longrightarrow \text{separating } \text{TYPE}('a) \ B \rangle$
unfolding *separating-def* **by** (*meson in-mono*)

lemma register-eqI: $\langle \text{separating TYPE('b::type)} A \implies \text{preregister } F \implies \text{preregister } G \implies (\bigwedge x. x \in A \implies F x = G x) \implies F = (G::- \Rightarrow 'b \text{ update}) \rangle$
unfolding separating-def by auto

lemma separating-tensor:

fixes $A :: \langle 'a::\text{type update set} \rangle$ **and** $B :: \langle 'b::\text{type update set} \rangle$
assumes $[simp]: \langle \text{separating TYPE('c::type)} A \rangle$
assumes $[simp]: \langle \text{separating TYPE('c)} B \rangle$
shows $\langle \text{separating TYPE('c)} \{a \otimes_u b \mid a \in A \wedge b \in B\} \rangle$
proof (*unfold separating-def, intro allI impI*)
fix $F G :: \langle ('a \times 'b) \text{ update} \Rightarrow 'c \text{ update} \rangle$
assume $[simp]: \langle \text{preregister } F \rangle \langle \text{preregister } G \rangle$
have $[simp]: \langle \text{preregister } (\lambda x. F (a \otimes_u x)) \rangle$ **for** a
using - $\langle \text{preregister } F \rangle$ **apply** (*rule comp-preregister[unfolded o-def]*)
by *simp*
have $[simp]: \langle \text{preregister } (\lambda x. G (a \otimes_u x)) \rangle$ **for** a
using - $\langle \text{preregister } G \rangle$ **apply** (*rule comp-preregister[unfolded o-def]*)
by *simp*
have $[simp]: \langle \text{preregister } (\lambda x. F (x \otimes_u b)) \rangle$ **for** b
using - $\langle \text{preregister } F \rangle$ **apply** (*rule comp-preregister[unfolded o-def]*)
by *simp*
have $[simp]: \langle \text{preregister } (\lambda x. G (x \otimes_u b)) \rangle$ **for** b
using - $\langle \text{preregister } G \rangle$ **apply** (*rule comp-preregister[unfolded o-def]*)
by *simp*
assume $\langle \forall x \in \{a \otimes_u b \mid a \in A \wedge b \in B\}. F x = G x \rangle$
then have $EQ: \langle F (a \otimes_u b) = G (a \otimes_u b) \rangle$ **if** $\langle a \in A \rangle$ **and** $\langle b \in B \rangle$ **for** $a \ b$
using *that by auto*
then have $\langle F (a \otimes_u b) = G (a \otimes_u b) \rangle$ **if** $\langle a \in A \rangle$ **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 $[simp]: \langle \text{register } F \rangle \langle \text{register } G \rangle \langle \text{register } H \rangle \langle \text{register } L \rangle$
shows $\langle (F \otimes_r G) \circ (H \otimes_r L) = (F \circ H) \otimes_r (G \circ L) \rangle$
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 $A :: \langle 'a::\text{type update set} \rangle$ **and** $B :: \langle 'b::\text{type update set} \rangle$
assumes $\langle \text{separating TYPE('c::type)} A \rangle$
assumes $\langle \text{separating TYPE('c)} B \rangle$
assumes $\langle C = \{a \otimes_u b \mid a \in A \wedge b \in B\} \rangle$
shows $\langle \text{separating TYPE('c)} C \rangle$
using *assms*
by (*simp add: separating-tensor*)

lemma tensor-extensionality3:

fixes $F G :: \langle ('a::\text{type} \times 'b::\text{type} \times 'c::\text{type}) \text{ update} \Rightarrow 'd::\text{type update} \rangle$
assumes $[simp]: \langle \text{register } F \rangle \langle \text{register } G \rangle$
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= $\{a \otimes_u b \otimes_u c \mid a \ b \ c. \text{True}\}$]*)
have $\langle \text{separating TYPE('d)} \{b \otimes_u c \mid b \ c. \text{True}\} \rangle$
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 $\langle (F; G) (a \otimes_u b) = (G b) *_u (F a) \rangle$
apply (*subst register-pair-apply*)
using *assms* **by** (*auto simp: compatible-def intro: register-preregister*)

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

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

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

lemma *compatible-register1*: \langle *compatible F G* \implies *register F* \rangle
by (*simp add: compatible-def*)

lemma *compatible-register2*: \langle *compatible F G* \implies *register G* \rangle
by (*simp add: compatible-def*)

lemma *pair-o-tensor*:
assumes *compatible A B* **and** [*simp*]: \langle *register C* \rangle **and** [*simp*]: \langle *register D* \rangle
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 $F :: 'a::\text{type update} \Rightarrow 'c::\text{type update}$ **and** $G :: 'b::\text{type update} \Rightarrow 'c::\text{type update}$
assumes *compatible F G*
shows *compatible* $(\lambda a. \text{Some } \otimes_u F a) (\lambda a. \text{Some } \otimes_u G a)$
using *assms* **apply** (*rule compatible-comp-inner[unfolded o-def]*)
by *simp*

lemma *compatible-tensor-id-update-right*[*simp*]:
fixes $F :: 'a::\text{type update} \Rightarrow 'c::\text{type update}$ **and** $G :: 'b::\text{type update} \Rightarrow 'c::\text{type update}$
assumes *compatible F G*
shows *compatible* $(\lambda a. F a \otimes_u \text{Some}) (\lambda a. G a \otimes_u \text{Some})$
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* $(\lambda a. F a \otimes_u \text{Some}) (\lambda a. \text{Some } \otimes_u G a)$
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* $(\lambda a. \text{Some } \otimes_u F a) (\lambda a. G a \otimes_u \text{Some})$
apply (*rule compatibleI*)
using *assms* **by** (*auto simp: tensor-update-mult*)

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

have [*simp*]: \langle *compatible (F o G) (F o H)* \rangle
apply (*rule compatible-comp-inner, simp*)
by *simp*

then have $[simp]: \langle register (F \circ G) \rangle \langle register (F \circ H) \rangle$
unfolding *compatible-def* **by** *auto*
from *assms* **have** $[simp]: \langle register G \rangle \langle register H \rangle$
unfolding *compatible-def* **by** *auto*
fix $a b$
show $\langle (F \circ G; F \circ H) (a \otimes_u b) = (F \circ (G; H)) (a \otimes_u b) \rangle$
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** $\langle Fst a = a \otimes_u Some \rangle$

definition *Snd* **where** $\langle Snd a = Some \otimes_u a \rangle$

lemma *register-Fst* $[simp]: \langle register Fst \rangle$
unfolding *Fst-def* **by** (*rule register-tensor-left*)

lemma *register-Snd* $[simp]: \langle register Snd \rangle$
unfolding *Snd-def* **by** (*rule register-tensor-right*)

lemma *compatible-Fst-Snd* $[simp]: \langle compatible Fst Snd \rangle$
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 $\langle swap = (Snd; Fst) \rangle$

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

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

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

lemma *register-swap* $[simp]: \langle register swap \rangle$
by (*simp add: swap-def*)

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

lemma *swap-o-swap* $[simp]: \langle swap \circ swap = id \rangle$
by (*metis swap-def compatible-Snd-Fst pair-Fst-Snd register-comp-pair register-swap swap-o-Fst swap-o-Snd*)

lemma *swap-swap* $[simp]: \langle swap (swap x) = x \rangle$
by (*simp add: pointfree-idE*)

lemma *inv-swap* $[simp]: \langle inv swap = swap \rangle$

by (meson inv-unique-comp swap-o-swap)

lemma *register-pair-Fst*:
assumes $\langle \text{compatible } F \ G \rangle$
shows $\langle (F; G) \circ \text{Fst} = F \rangle$
using *assms* **by** (auto intro!: ext simp: Fst-def register-pair-apply compatible-register2)

lemma *register-pair-Snd*:
assumes $\langle \text{compatible } F \ G \rangle$
shows $\langle (F; G) \circ \text{Snd} = G \rangle$
using *assms* **by** (auto intro!: ext simp: Snd-def register-pair-apply compatible-register1)

lemma *register-Fst-register-Snd[simp]*:
assumes $\langle \text{register } F \rangle$
shows $\langle (F \circ \text{Fst}; F \circ \text{Snd}) = F \rangle$
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 $\langle \text{register } F \rangle$
shows $\langle (F \circ \text{Snd}; F \circ \text{Fst}) = F \circ \text{swap} \rangle$
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]: $\langle \text{register } F \rangle \langle \text{register } G \rangle \langle \text{register } H \rangle$

using *assms* *compatible-def* **by** auto

then have [simp]: $\langle \text{preregister } F \rangle \langle \text{preregister } G \rangle \langle \text{preregister } H \rangle$

using *register-preregister* **by** blast+

have [simp]: $\langle \text{preregister } (\lambda a. (F; G) \ a \ *_{\text{u}} \ z) \rangle$ **for** z

apply (rule comp-preregister[unfolded o-def, of $\langle (F; G) \rangle$])

by *simp-all*

have [simp]: $\langle \text{preregister } (\lambda a. z \ *_{\text{u}} \ (F; G) \ a) \rangle$ **for** z

apply (rule comp-preregister[unfolded o-def, of $\langle (F; G) \rangle$])

by *simp-all*

have $(F; G) \ (f \otimes_{\text{u}} g) \ *_{\text{u}} \ H \ h = H \ h \ *_{\text{u}} \ (F; G) \ (f \otimes_{\text{u}} g)$ **for** $f \ g \ h$

proof –

have $FH: F \ f \ *_{\text{u}} \ H \ h = H \ h \ *_{\text{u}} \ F \ f$

using *assms* *compatible-def* **by** *metis*

have $GH: G \ g \ *_{\text{u}} \ H \ h = H \ h \ *_{\text{u}} \ G \ g$

using *assms* *compatible-def* **by** *metis*

have $\langle (F; G) \ (f \otimes_{\text{u}} g) \ *_{\text{u}} \ (H \ h) = F \ f \ *_{\text{u}} \ G \ g \ *_{\text{u}} \ H \ h \rangle$

using $\langle \text{compatible } F \ G \rangle$ **by** (subst register-pair-apply, auto)

also have $\langle \dots = H \ h \ *_{\text{u}} \ F \ f \ *_{\text{u}} \ G \ g \rangle$

using $FH \ GH$ **by** (*metis* comp-update-assoc)

also have $\langle \dots = H \ h \ *_{\text{u}} \ (F; G) \ (f \otimes_{\text{u}} g) \rangle$

using $\langle \text{compatible } F \ G \rangle$ **by** (subst register-pair-apply, auto simp: comp-update-assoc)

finally show *?thesis*

by –

qed

then show $(F; G) \ fg \ *_{\text{u}} \ (H \ h) = (H \ h) \ *_{\text{u}} \ (F; G) \ fg$ **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) o swap)⟩
    by simp
  show ⟨preregister (B; A)⟩
    by (metis (no-types, lifting) assms compatible-sym register-preregister pair-is-register)
  show ⟨((A; B) o 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 ΦI: ⟨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  $\Phi 2$ :  $\langle \Phi \ c \ d \ \sigma = (F' \otimes_r G') (c \otimes_u d) *_u (F \otimes_r G) \ \sigma \rangle$  for  $c \ d \ \sigma$ 
  apply (rule-tac fun-cong[of - -  $\sigma$ ])
  apply (rule tensor-extensionality)
  by auto
from  $\Phi 1 \ \Phi 2$  have  $\langle (F \otimes_r G) \ \sigma *_u (F' \otimes_r G') \ \tau = (F' \otimes_r G') \ \tau *_u (F \otimes_r G) \ \sigma \rangle$  for  $\tau \ \sigma$ 
  apply (rule-tac fun-cong[of - -  $\tau$ ])
  apply (rule tensor-extensionality)
  by auto
then show ?thesis
  apply (rule compatibleI[rotated - I])
  by auto
qed

```

6.8 Associativity of the tensor product

definition $assoc :: \langle ('a::type \times 'b::type) \times 'c::type \rangle \text{ update} \Rightarrow \langle 'a \times ('b \times 'c) \rangle \text{ update}$ where
 $\langle assoc = ((Fst; Snd \ o \ Fst); Snd \ o \ Snd) \rangle$

lemma $assoc\text{-is-hom}[simp]$: $\langle \text{preregister } assoc \rangle$
 by (auto simp: assoc-def)

lemma $assoc\text{-apply}[simp]$: $\langle assoc \ ((a \otimes_u b) \otimes_u c) = (a \otimes_u (b \otimes_u c)) \rangle$
 by (auto simp: assoc-def register-pair-apply Fst-def Snd-def tensor-update-mult)

definition $assoc' :: \langle 'a \times ('b \times 'c) \rangle \text{ update} \Rightarrow \langle ('a::type \times 'b::type) \times 'c::type \rangle \text{ update}$ where
 $\langle assoc' = (Fst \ o \ Fst; (Fst \ o \ Snd; Snd)) \rangle$

lemma $assoc'\text{-is-hom}[simp]$: $\langle \text{preregister } assoc' \rangle$
 by (auto simp: assoc'-def)

lemma $assoc'\text{-apply}[simp]$: $\langle assoc' \ (a \otimes_u (b \otimes_u c)) = ((a \otimes_u b) \otimes_u c) \rangle$
 by (auto simp: assoc'-def register-pair-apply Fst-def Snd-def tensor-update-mult)

lemma $register\text{-assoc}[simp]$: $\langle \text{register } assoc \rangle$
 unfolding assoc-def
 by force

lemma $register\text{-assoc}'[simp]$: $\langle \text{register } assoc' \rangle$
 unfolding assoc'-def
 by force

lemma $pair\text{-o-}assoc[simp]$:
 assumes [simp]: $\langle \text{compatible } F \ G \rangle \langle \text{compatible } G \ H \rangle \langle \text{compatible } F \ H \rangle$
 shows $\langle (F; (G; H)) \circ assoc = ((F; G); H) \rangle$
proof (rule tensor-extensionality3')
 show $\langle \text{register } ((F; (G; H)) \circ assoc) \rangle$
 by simp
 show $\langle \text{register } ((F; G); H) \rangle$
 by simp
 show $\langle ((F; (G; H)) \circ assoc) \ ((f \otimes_u g) \otimes_u h) = ((F; G); H) \ ((f \otimes_u g) \otimes_u h) \rangle$ for $f \ g \ h$
 by (simp add: register-pair-apply assoc-apply comp-update-assoc)
 qed

lemma $pair\text{-o-}assoc'[simp]$:
 assumes [simp]: $\langle \text{compatible } F \ G \rangle \langle \text{compatible } G \ H \rangle \langle \text{compatible } F \ H \rangle$
 shows $\langle ((F; G); H) \circ assoc' = (F; (G; H)) \rangle$
proof (rule tensor-extensionality3)
 show $\langle \text{register } (((F; G); H) \circ assoc') \rangle$
 by simp
 show $\langle \text{register } (F; (G; H)) \rangle$
 by simp
 show $\langle (((F; G); H) \circ assoc') \ (f \otimes_u g \otimes_u h) = (F; (G; H)) \ (f \otimes_u g \otimes_u h) \rangle$ for $f \ g \ h$

by (simp add: register-pair-apply assoc'-apply comp-update-assoc)
qed

lemma assoc'-o-assoc[simp]: $\langle \text{assoc}' \circ \text{assoc} = \text{id} \rangle$
 apply (rule tensor-extensionality3')
 by auto

lemma assoc'-assoc[simp]: $\langle \text{assoc}' (\text{assoc } x) = x \rangle$
 by (simp add: pointfree-idE)

lemma assoc-o-assoc'[simp]: $\langle \text{assoc} \circ \text{assoc}' = \text{id} \rangle$
 apply (rule tensor-extensionality3)
 by auto

lemma assoc-assoc'[simp]: $\langle \text{assoc} (\text{assoc}' x) = x \rangle$
 by (simp add: pointfree-idE)

lemma inv-assoc[simp]: $\langle \text{inv } \text{assoc} = \text{assoc}' \rangle$
 using assoc'-o-assoc assoc-o-assoc' inv-unique-comp by blast

lemma inv-assoc'[simp]: $\langle \text{inv } \text{assoc}' = \text{assoc} \rangle$
 by (simp add: inv-equality)

lemma [simp]: $\langle \text{bij } \text{assoc} \rangle$
 using assoc'-o-assoc assoc-o-assoc' o-bij by blast

lemma [simp]: $\langle \text{bij } \text{assoc}' \rangle$
 using assoc'-o-assoc assoc-o-assoc' o-bij by blast

6.9 Iso-registers

definition $\langle \text{iso-register } F \iff \text{register } F \wedge (\exists G. \text{register } G \wedge F \circ G = \text{id} \wedge G \circ F = \text{id}) \rangle$
 for $F :: \langle \text{--}::\text{type update} \Rightarrow \text{--}::\text{type update} \rangle$

lemma iso-registerI:
 assumes $\langle \text{register } F \rangle \langle \text{register } G \rangle \langle F \circ G = \text{id} \rangle \langle G \circ F = \text{id} \rangle$
 shows $\langle \text{iso-register } F \rangle$
 using assms(1) assms(2) assms(3) assms(4) iso-register-def by blast

lemma iso-register-inv: $\langle \text{iso-register } F \implies \text{iso-register } (\text{inv } F) \rangle$
 by (metis inv-unique-comp iso-register-def)

lemma iso-register-inv-comp1: $\langle \text{iso-register } F \implies \text{inv } F \circ F = \text{id} \rangle$
 using inv-unique-comp iso-register-def by blast

lemma iso-register-inv-comp2: $\langle \text{iso-register } F \implies F \circ \text{inv } F = \text{id} \rangle$
 using inv-unique-comp iso-register-def by blast

lemma iso-register-id[simp]: $\langle \text{iso-register } \text{id} \rangle$
 by (simp add: iso-register-def)

lemma iso-register-is-register: $\langle \text{iso-register } F \implies \text{register } F \rangle$
 using iso-register-def by blast

lemma iso-register-comp[simp]:
 assumes [simp]: $\langle \text{iso-register } F \rangle \langle \text{iso-register } G \rangle$
 shows $\langle \text{iso-register } (F \circ G) \rangle$

proof –

from assms obtain $F' G'$ where [simp]: $\langle \text{register } F' \rangle \langle \text{register } G' \rangle \langle F \circ F' = \text{id} \rangle \langle F' \circ F = \text{id} \rangle$
 $\langle G \circ G' = \text{id} \rangle \langle G' \circ G = \text{id} \rangle$
 by (meson iso-register-def)
 show ?thesis

```

apply (rule iso-registerI[where  $G = \langle G' \circ F' \rangle$ ])
  apply (auto simp: register-tensor-is-register iso-register-is-register register-tensor-distrib)
  apply (metis  $\langle F \circ F' = id \rangle \langle G \circ G' = id \rangle$  fcomp-assoc fcomp-comp id-fcomp)
  by (metis (no-types, lifting)  $\langle F \circ F' = id \rangle \langle F' \circ F = id \rangle \langle G' \circ G = id \rangle$  fun.map-comp inj-iff inv-unique-comp
o-inv-o-cancel)
qed

```

```

lemma iso-register-tensor-is-iso-register[simp]:
  assumes [simp]:  $\langle iso-register\ F \rangle \langle iso-register\ G \rangle$ 
  shows  $\langle iso-register\ (F \otimes_r G) \rangle$ 
proof –
  from assms obtain  $F' G'$  where [simp]:  $\langle register\ F' \rangle \langle register\ G' \rangle \langle F \circ F' = id \rangle \langle F' \circ F = id \rangle$ 
 $\langle G \circ G' = id \rangle \langle G' \circ G = id \rangle$ 
  by (meson iso-register-def)
  show ?thesis
  apply (rule iso-registerI[where  $G = \langle F' \otimes_r G' \rangle$ ])
  by (auto simp: register-tensor-is-register iso-register-is-register register-tensor-distrib)
qed

```

```

lemma iso-register-bij:  $\langle iso-register\ F \implies bij\ F \rangle$ 
using iso-register-def o-bij by auto

```

```

lemma inv-register-tensor[simp]:
  assumes [simp]:  $\langle iso-register\ F \rangle \langle iso-register\ G \rangle$ 
  shows  $\langle inv\ (F \otimes_r G) = inv\ F \otimes_r inv\ G \rangle$ 
  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]:  $\langle iso-register\ swap \rangle$ 
apply (rule iso-registerI[of - swap])
by auto

```

```

lemma iso-register-assoc[simp]:  $\langle iso-register\ assoc \rangle$ 
apply (rule iso-registerI[of - assoc])
by auto

```

```

lemma iso-register-assoc'[simp]:  $\langle iso-register\ assoc' \rangle$ 
apply (rule iso-registerI[of - assoc])
by auto

```

```

definition  $\langle equivalent-registers\ F\ G \iff (register\ F \wedge (\exists I. iso-register\ I \wedge F \circ I = G)) \rangle$ 
for  $F\ G :: \langle \cdot \rangle :: type\ update \Rightarrow \langle \cdot \rangle :: type\ update \rangle$ 

```

```

lemma iso-register-equivalent-id[simp]:  $\langle equivalent-registers\ id\ F \iff iso-register\ F \rangle$ 
by (simp add: equivalent-registers-def)

```

```

lemma equivalent-registersI:
  assumes  $\langle register\ F \rangle$ 
  assumes  $\langle iso-register\ I \rangle$ 
  assumes  $\langle F \circ I = G \rangle$ 
  shows  $\langle equivalent-registers\ F\ G \rangle$ 
using assms unfolding equivalent-registers-def by blast

```

```

lemma equivalent-registers-register-left:  $\langle equivalent-registers\ F\ G \implies register\ F \rangle$ 
using equivalent-registers-def by auto

```

```

lemma equivalent-registers-register-right:  $\langle register\ G \rangle$  if  $\langle equivalent-registers\ F\ G \rangle$ 
by (metis equivalent-registers-def iso-register-def register-comp that)

```

```

lemma equivalent-registers-sym:

```

assumes $\langle \text{equivalent-registers } F \ G \rangle$
shows $\langle \text{equivalent-registers } G \ F \rangle$
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 $\langle \text{equivalent-registers } F \ G \rangle$ **and** $\langle \text{equivalent-registers } G \ H \rangle$
shows $\langle \text{equivalent-registers } F \ H \rangle$
proof –
from *assms* **have** [*simp*]: $\langle \text{register } F \rangle \langle \text{register } G \rangle$
by (*auto simp: equivalent-registers-def*)
from *assms*(1) **obtain** *I* **where** [*simp*]: $\langle \text{iso-register } I \rangle$ **and** $\langle F \circ I = G \rangle$
using *equivalent-registers-def* **by** *blast*
from *assms*(2) **obtain** *J* **where** [*simp*]: $\langle \text{iso-register } J \rangle$ **and** $\langle G \circ J = H \rangle$
using *equivalent-registers-def* **by** *blast*
have $\langle \text{register } F \rangle$
by (*auto simp: equivalent-registers-def*)
moreover **have** $\langle \text{iso-register } (I \circ J) \rangle$
using $\langle \text{iso-register } I \rangle \langle \text{iso-register } J \rangle$ *iso-register-comp* **by** *blast*
moreover **have** $\langle F \circ (I \circ J) = H \rangle$
by (*simp add: \langle F \circ I = G \rangle \langle G \circ J = H \rangle o-assoc*)
ultimately **show** *?thesis*
by (*rule equivalent-registersI*)
qed

lemma *equivalent-registers-assoc*[*simp*]:
assumes [*simp*]: $\langle \text{compatible } F \ G \rangle \langle \text{compatible } F \ H \rangle \langle \text{compatible } G \ H \rangle$
shows $\langle \text{equivalent-registers } (F; (G; H)) \ ((F; G); H) \rangle$
apply (*rule equivalent-registersI*[**where** *I=assoc*])
by *auto*

lemma *equivalent-registers-pair-right*:
assumes [*simp*]: $\langle \text{compatible } F \ G \rangle$
assumes *eq*: $\langle \text{equivalent-registers } G \ H \rangle$
shows $\langle \text{equivalent-registers } (F; G) \ (F; H) \rangle$
proof –
from *eq* **obtain** *I* **where** [*simp*]: $\langle \text{iso-register } I \rangle$ **and** $\langle G \circ I = H \rangle$
by (*metis equivalent-registers-def*)
then **have** $*$: $\langle (F; G) \circ (\text{id} \otimes_r I) = (F; H) \rangle$
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*]: $\langle \text{compatible } F \ G \rangle$
assumes *eq*: $\langle \text{equivalent-registers } F \ H \rangle$
shows $\langle \text{equivalent-registers } (F; G) \ (H; G) \rangle$
proof –
from *eq* **obtain** *I* **where** [*simp*]: $\langle \text{iso-register } I \rangle$ **and** $\langle F \circ I = H \rangle$
by (*metis equivalent-registers-def*)
then **have** $*$: $\langle (F; G) \circ (I \otimes_r \text{id}) = (H; G) \rangle$
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 $\langle \text{register } H \rangle$
assumes $\langle \text{equivalent-registers } F \ G \rangle$

shows $\langle \text{equivalent-registers } (H \circ F) (H \circ G) \rangle$
by $(\text{metis } (\text{no-types, lifting}) \text{ assms}(1) \text{ assms}(2) \text{ comp-assoc equivalent-registers-def register-comp})$

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) =  $\langle$ 
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  $\langle$ compatible $\rangle$ , -) $ x $ y => (x, SOME y)
    | Const(const-name  $\langle$ register $\rangle$ , -) $ 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 or else 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]:  $\langle$  ^ 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]:  $\langle$  ^ 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 \langle compatible - \rightarrow \rangle] asm-rl[of \langle iso-register \rightarrow \rangle] asm-rl[of \langle register \rightarrow \rangle] 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 \langle

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 ⊗_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

```

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 (T1)
no-notation m-inv (inv1 - [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 (ctx,namepos) => let
  val facts = Proof-Context.facts-of ctx
  val fullname = Facts.check (Context.Proof ctx) facts namepos
  val (markup, shortname) = Proof-Context.markup-extern-fact ctx 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 <math>\longleftrightarrow P (0::bit) \wedge P 1</math>
definition enum-ex-bit P <math>\longleftrightarrow P (0::bit) \vee P 1</math>
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]: <math>(*_V) id-cblinfun = id</math>
by auto

```

Overriding

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

```

definition sandwich where <math>sandwich a b = a o_{CL} b o_{CL} a^*</math>

```

```

lemma clinear-sandwich[simp]: <math>clinear (sandwich a)</math>
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]: <math>sandwich id-cblinfun = id</math>
by (auto simp: sandwich-def)

```

```

lemma mat-of-cblinfun-sandwich:
  fixes a :: (onb-enum, onb-enum) cblinfun
  shows <math>mat-of-cblinfun (sandwich a b) = (let a' = mat-of-cblinfun a in a' * mat-of-cblinfun b * mat-adjoint a')</math>
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 $\langle a\ o_{CL}\ b = c \rangle$
shows $\langle a\ o_{CL}\ b = c \rangle$
and $\langle a\ o_{CL}\ (b\ o_{CL}\ d) = c\ o_{CL}\ d \rangle$
and $\langle a\ *_{S}\ (b\ *_{S}\ S) = c\ *_{S}\ S \rangle$
and $\langle a\ *_{V}\ (b\ *_{V}\ x) = c\ *_{V}\ x \rangle$
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* $f\ (x_1, x_2, \dots, x_n)$ expands to the conjunction of all $f\ x_i\ x_j$ with $i \neq j$.
- *each* $f\ (x_1, x_2, \dots, x_n)$ expands to the conjunction of all $f\ x_i$.

syntax *-mutually* $:: 'a \Rightarrow args \Rightarrow 'b\ (mutually\ -\ '(-))$
syntax *-mutually2* $:: 'a \Rightarrow 'b \Rightarrow args \Rightarrow args \Rightarrow 'c$

translations *mutually* $f\ (x) \Rightarrow CONST\ True$
translations *mutually* $f\ (-args\ x\ y) \Rightarrow f\ x\ y \wedge f\ y\ x$
translations *mutually* $f\ (-args\ x\ (-args\ x'\ xs)) \Rightarrow -mutually2\ f\ x\ (-args\ x'\ xs)\ (-args\ x'\ xs)$
translations *-mutually2* $f\ x\ y\ zs \Rightarrow f\ x\ y \wedge f\ y\ x \wedge -mutually\ f\ zs$
translations *-mutually2* $f\ x\ (-args\ y\ ys)\ zs \Rightarrow f\ x\ y \wedge f\ y\ x \wedge -mutually2\ f\ x\ ys\ zs$

syntax *-each* $:: 'a \Rightarrow args \Rightarrow 'b\ (each\ -\ '(-))$
translations *each* $f\ (x) \Rightarrow f\ x$
translations *-each* $f\ (-args\ x\ xs) \Rightarrow f\ x \wedge -each\ f\ xs$

lemma *enum-inj*:
assumes $i < CARD('a)$ **and** $j < CARD('a)$
shows $(Enum.enum\ !\ i :: 'a::enum) = Enum.enum\ !\ j \longleftrightarrow i = j$
using *inj-on-nth[OF enum-distinct, where I = $\langle \dots < CARD('a) \rangle$]*
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 $\langle inj\ f \rangle$
assumes $\langle x = f\ y \rangle$
shows $\langle inv\ f\ x = y \rangle$
by (*simp add: assms(1) assms(2)*)

instantiation *prod* $:: (default, default)\ default\ \mathbf{begin}$
definition $\langle default-prod = (default, default) \rangle$
instance..
end

instance *bit* $:: default..$

lemma *surj-from-comp*:
assumes $\langle surj\ (g\ o\ f) \rangle$
assumes $\langle inj\ g \rangle$
shows $\langle surj\ f \rangle$
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*: $\langle (\exists x y. Q x y) \longleftrightarrow (\exists z. Q (fst z) (snd z)) \rangle$
 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*]: $\langle register-from-getter-setter (getter F) (setter F) = F \rangle$
 if $\langle register F \rangle$
 by (metis *getter-of-register-from-getter-setter register-def setter-of-register-from-getter-setter that*)

lemma *valid-getter-setter-getter-setter*[*simp*]: $\langle valid-getter-setter (getter F) (setter F) \rangle$ if $\langle register F \rangle$
 by (metis *getter-of-register-from-getter-setter register-def setter-of-register-from-getter-setter that*)

lemma *register-register-from-getter-setter*[*simp*]: $\langle register (register-from-getter-setter g s) \rangle$ if $\langle valid-getter-setter g s \rangle$
 using *register-def that* by *blast*

definition $\langle total-fun f = (\forall x. f x \neq None) \rangle$

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

lemma *register-apply*:
 assumes $\langle register F \rangle$
 shows $\langle Some o register-apply F a = F (Some o a) \rangle$
proof –
 have $\langle total-fun (F (Some o a)) \rangle$
 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 $\langle preregister F \rangle$
 shows $\langle F Map.empty = Map.empty \rangle$
 using *assms unfolding preregister-def* by *auto*

lemma *compatible-setter*:
 fixes $F :: \langle ('a, 'c) preregister \rangle$ and $G :: \langle ('b, 'c) preregister \rangle$
 assumes [*simp*]: $\langle register F \rangle \langle register G \rangle$
 shows $\langle compatible F G \longleftrightarrow (\forall a b. setter F a o setter G b = setter G b o setter F a) \rangle$

proof (*intro allI iffI*)
 fix $a b$
 assume $\langle compatible F G \rangle$
 then show $\langle setter F a o setter G b = setter G b o setter F a \rangle$
 apply (*rule-tac compatible-setter*)
 unfolding *compatible-def* by *auto*

next
 assume *commute*[*rule-format, THEN fun-cong, unfolded o-def*]: $\langle \forall a b. setter F a o setter G b = setter G b o setter F a \rangle$
 have $\langle valid-getter-setter (getter F) (setter F) \rangle$
 by *auto*

then have $\langle F a \circ_m G b = G b \circ_m F a \rangle$ **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 $\langle \text{compatible } F G \rangle$
unfolding *compatible-def* **by** *auto*
qed

lemma *register-from-getter-setter-compatibleI*[*intro*]:
assumes [*simp*]: $\langle \text{valid-getter-setter } g s \rangle \langle \text{valid-getter-setter } g' s' \rangle$
assumes $\langle \bigwedge x y m. s x (s' y m) = s' y (s x m) \rangle$
shows $\langle \text{compatible (register-from-getter-setter } g s) (\text{register-from-getter-setter } g' s') \rangle$
apply (*subst compatible-setter*)
using *assms by auto*

lemma *separating-update1*:
 $\langle \text{separating } \text{TYPE}(-) \{ \text{update1 } x y \mid x y. \text{True} \} \rangle$
by (*smt (verit) mem-Collect-eq separating-def update1-extensionality*)

definition *permutation-register* ($p :: 'b \Rightarrow 'a$) = *register-from-getter-setter* p ($\lambda a -. \text{inv } p a$)

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*: $\langle \text{bij } p \implies \text{getter (permutation-register } p) = p \rangle$
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*: $\langle \text{bij } p \implies \text{setter (permutation-register } p) a m = \text{inv } p a \rangle$
by (*metis bij-inv-eq-iff getter-permutation-register permutation-register-register valid-getter-setter-def valid-getter-setter-getter-setter*)

definition *empty-var* :: $\langle 'a :: \{ \text{CARD-1} \} \text{update} \Rightarrow 'b \text{update} \rangle$ **where**
empty-var = *register-from-getter-setter* ($\lambda -. \text{undefined}$) ($\lambda -. m. m$)

lemma *valid-empty-var*[*simp*]: $\langle \text{valid-getter-setter } (\lambda -. (\text{undefined} :: \text{CARD-1})) (\lambda -. m. m) \rangle$
by (*simp add: valid-getter-setter-def*)

lemma *register-empty-var*[*simp*]: $\langle \text{register empty-var} \rangle$
using *empty-var-def register-def valid-empty-var* **by** *blast*

lemma *getter-empty-var*[*simp*]: $\langle \text{getter empty-var } m = \text{undefined} \rangle$
by (*rule everything-the-same*)

lemma *setter-empty-var*[*simp*]: $\langle \text{setter empty-var } a m = m \rangle$
by (*simp add: empty-var-def setter-of-register-from-getter-setter*)

lemma *empty-var-compatible*[*simp*]: $\langle \text{compatible empty-var } X \rangle$ **if** [*simp*]: $\langle \text{register } X \rangle$
apply (*subst compatible-setter*) **by** *auto*

lemma *empty-var-compatible'*[*simp*]: $\langle \text{register } X \implies \text{compatible } X \text{ empty-var} \rangle$
using *compatible-sym empty-var-compatible* **by** *blast*

```

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

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

lemma validX[simp]: ⟨valid-getter-setter x (λa 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 (λa 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 (inv1 - [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 (λb. 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-ell2*: *clinear* ($\lambda a.$ *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* :: $\langle ('a$ *ell2*, $'b::$ finite *ell2*) *cblinfun* \Rightarrow ($'c$ *ell2*, $'d::$ finite *ell2*) *cblinfun*
 \Rightarrow ($('a \times 'c)$ *ell2*, $('b \times 'd)$ *ell2*) *cblinfun* (infixr \otimes_o 70) **where**
 \langle *tensor-op* M N = (SOME $P.$ $\forall a$ $c.$ $P *_{\mathcal{V}}$ (*ket* (a,c)))
 = *tensor-ell2* ($M *_{\mathcal{V}}$ *ket* a) ($N *_{\mathcal{V}}$ *ket* c) \rangle

lemma *tensor-op-ket*:

fixes $a :: \langle 'a::$ finite \rangle and $b :: \langle 'b \rangle$ and $c :: \langle 'c::$ finite \rangle and $d :: \langle 'd \rangle$
 shows \langle *tensor-op* M $N *_{\mathcal{V}}$ (*ket* (a,c)) = *tensor-ell2* ($M *_{\mathcal{V}}$ *ket* a) ($N *_{\mathcal{V}}$ *ket* c) \rangle

proof –

define $S :: \langle ('a \times 'c)$ *ell2* *set* \rangle **where** $S =$ *ket* ' *UNIV*
 define φ **where** $\langle \varphi = (\lambda(a,c).$ *tensor-ell2* ($M *_{\mathcal{V}}$ *ket* a) ($N *_{\mathcal{V}}$ *ket* c)) \rangle
 define φ' **where** $\langle \varphi' = \varphi \circ$ *inv* *ket* \rangle

have *def*: \langle *tensor-op* M N = (SOME $P.$ $\forall a$ $c.$ $P *_{\mathcal{V}}$ (*ket* (a,c)) = φ (a,c)) \rangle
 unfolding *tensor-op-def* φ -*def* by auto

have \langle *cindependent* S \rangle
 using *S-def* *cindependent-ket* by blast
 moreover **have** \langle *cspan* $S =$ *UNIV* \rangle
 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** $\exists P. \forall x \in S. P *_{\mathcal{V}}$ $x = \varphi' x$
 unfolding *cblinfun-extension-exists-def* by auto
 then **have** *ex*: $\langle \exists P. \forall a$ $c. P *_{\mathcal{V}}$ *ket* (a,c) = φ (a,c) \rangle
 by (*metis* *S-def* φ' -*def* *comp-eq-dest-lhs inj-ket inv-f-f rangeI*)

then have \langle *tensor-op* M $N *_{\mathcal{V}}$ (*ket* (a,c)) = φ (a,c) \rangle
 unfolding *def* apply (rule *someI2-ex*[**where** $P = \langle \lambda P. \forall a$ $c. P *_{\mathcal{V}}$ (*ket* (a,c)) = φ (a,c)]])
 by auto

then show ?*thesis*
 unfolding φ -*def* by auto

qed

lemma *tensor-op-ell2*: *tensor-op* A $B *_{\mathcal{V}}$ *tensor-ell2* ψ φ = *tensor-ell2* ($A *_{\mathcal{V}}$ ψ) ($B *_{\mathcal{V}}$ φ)

proof –

have 1: \langle *clinear* ($\lambda a.$ *tensor-op* A $B *_{\mathcal{V}}$ *tensor-ell2* a (*ket* b)) \rangle **for** b
 by (auto intro!: *clinearI* simp: *tensor-ell2-add1* *tensor-ell2-scaleC1* *cblinfun.add-right*)
 have 2: \langle *clinear* ($\lambda a.$ *tensor-ell2* ($A *_{\mathcal{V}}$ a) ($B *_{\mathcal{V}}$ *ket* b)) \rangle **for** b
 by (auto intro!: *clinearI* simp: *tensor-ell2-add1* *tensor-ell2-scaleC1* *cblinfun.add-right*)
 have 3: \langle *clinear* ($\lambda a.$ *tensor-op* A $B *_{\mathcal{V}}$ *tensor-ell2* ψ a) \rangle
 by (auto intro!: *clinearI* simp: *tensor-ell2-add2* *tensor-ell2-scaleC2* *cblinfun.add-right*)
 have 4: \langle *clinear* ($\lambda a.$ *tensor-ell2* ($A *_{\mathcal{V}}$ ψ) ($B *_{\mathcal{V}}$ a)) \rangle
 by (auto intro!: *clinearI* simp: *tensor-ell2-add2* *tensor-ell2-scaleC2* *cblinfun.add-right*)

have *eq-ket-ket*: \langle *tensor-op* A $B *_{\mathcal{V}}$ *tensor-ell2* (*ket* a) (*ket* b) = *tensor-ell2* ($A *_{\mathcal{V}}$ *ket* a) ($B *_{\mathcal{V}}$ *ket* b) \rangle **for** a b
 by (simp add: *tensor-op-ket*)

have *eq-ket*: \langle *tensor-op* A $B *_{\mathcal{V}}$ *tensor-ell2* ψ (*ket* b) = *tensor-ell2* ($A *_{\mathcal{V}}$ ψ) ($B *_{\mathcal{V}}$ *ket* b) \rangle **for** b
 apply (rule *fun-cong*[**where** $x = \psi$])
 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) oCL (tensor-op c d) = tensor-op (a oCL c) (b oCL 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:  $\langle$ cbilinear (tensor-op :: 'a::finite ell2  $\Rightarrow_{CL}$  'b::finite ell2
 $\Rightarrow$  'c::finite ell2  $\Rightarrow_{CL}$  'd::finite ell2  $\Rightarrow$  -) $\rangle$ 

```

proof –

```

have  $\langle$ clinear ( $\lambda$ b::'c ell2  $\Rightarrow_{CL}$  'd ell2. tensor-op a b) $\rangle$  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  $\langle$ clinear ( $\lambda$ a::'a::finite ell2  $\Rightarrow_{CL}$  'b::finite ell2. tensor-op a b) $\rangle$  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:  $\langle$ tensor-op (butterket i j) (butterket k l) = butterket (i,k) (j,l) $\rangle$ 
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:  $\langle$ cspan {tensor-op (butterket i j) (butterket k l) | i (j::-::finite) k (l::-::finite). True} =
UNIV $\rangle$ 
  unfolding tensor-butter
  apply (subst cspan-butterfly-ket[symmetric])
  by (metis surj-pair)

```

```

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

```

lemma tensor-extensionality:

```

fixes F G ::  $\langle$ ((('a::finite  $\times$  'b::finite) ell2)  $\Rightarrow_{CL}$  (('c::finite  $\times$  'd::finite) ell2))  $\Rightarrow$  'e::complex-vector $\rangle$ 
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  $\langle$ clinear F $\rangle$  and  $\langle$ clinear G $\rangle$ 

```

```

  using assms by (simp-all add: cbilinear-def)

```

```

show  $\langle$ x  $\in$  cspan {tensor-op (butterket i j) (butterket k l) | i j k l. True} $\rangle$ 

```

for $x :: \langle ('a \times 'b) \text{ ell2} \Rightarrow_{CL} ('c \times 'd) \text{ ell2} \rangle$
using *cspan-tensor-op* **by** *auto*
show $\langle F x = G x \rangle$ **if** $\langle x \in \{ \text{tensor-op } (\text{butterket } i \ j) \ (\text{butterket } k \ l) \mid i \ j \ k \ l. \text{ True} \} \rangle$ **for** x
using *that* **by** $(\text{auto } \text{simp}: \text{tensor-eq})$
qed

lemma *tensor-id[simp]*: $\langle \text{tensor-op id-cblinfun id-cblinfun} = \text{id-cblinfun} \rangle$
apply $(\text{rule } \text{equal-ket}, \text{rename-tac } x, \text{case-tac } x)$
by $(\text{simp } \text{flip}: \text{tensor-ell2-ket } \text{add}: \text{tensor-op-ell2})$

lemma *tensor-op-adjoint*: $\langle (\text{tensor-op } a \ b)^* = \text{tensor-op } (a^*) \ (b^*) \rangle$
apply $(\text{rule } \text{cinner-ket-adjointI}[\text{symmetric}])$
apply $(\text{auto } \text{simp } \text{flip}: \text{tensor-ell2-ket } \text{simp}: \text{tensor-op-ell2})$
by $(\text{simp } \text{add}: \text{cinner-adj-left})$

lemma *tensor-butterfly[simp]*: $\text{tensor-op } (\text{butterfly } \psi \ \psi') \ (\text{butterfly } \varphi \ \varphi') = \text{butterfly } (\text{tensor-ell2 } \psi \ \varphi) \ (\text{tensor-ell2 } \psi' \ \varphi')$
apply $(\text{rule } \text{equal-ket}, \text{rename-tac } x, \text{case-tac } x)$
by $(\text{simp } \text{flip}: \text{tensor-ell2-ket } \text{add}: \text{tensor-op-ell2 } \text{butterfly-def} \ \text{cblinfun-apply-cblinfun-compose } \text{tensor-ell2-scaleC1 } \text{tensor-ell2-scaleC2})$

definition *tensor-lift* :: $\langle (('a1::\text{finite ell2} \Rightarrow_{CL} 'a2::\text{finite ell2}) \Rightarrow ('b1::\text{finite ell2} \Rightarrow_{CL} 'b2::\text{finite ell2}) \Rightarrow 'c) \Rightarrow (('a1 \times 'b1) \text{ ell2} \Rightarrow_{CL} ('a2 \times 'b2) \text{ ell2}) \Rightarrow 'c::\text{complex-vector} \rangle$ **where**
 $\text{tensor-lift } F2 = (\text{SOME } G. \text{clinear } G \wedge (\forall a \ b. G (\text{tensor-op } a \ b) = F2 \ a \ b))$

lemma
fixes $F2 :: 'a::\text{finite ell2} \Rightarrow_{CL} 'b::\text{finite ell2} \Rightarrow 'c::\text{finite ell2} \Rightarrow_{CL} 'd::\text{finite ell2} \Rightarrow 'e::\text{complex-normed-vector}$
assumes *cbilinear* $F2$
shows *tensor-lift-clinear*: *clinear* $(\text{tensor-lift } F2)$
and *tensor-lift-correct*: $\langle (\lambda a \ b. \text{tensor-lift } F2 (\text{tensor-op } a \ b)) = F2 \rangle$

proof –

define $F2' \ t4 \ \varphi$ **where**
 $\langle F2' = \text{tensor-lift } F2 \rangle$ **and**
 $\langle t4 = (\lambda(i,j,k,l). \text{tensor-op } (\text{butterket } i \ j) \ (\text{butterket } k \ l)) \rangle$ **and**
 $\langle \varphi \ m = (\text{let } (i,j,k,l) = \text{inv } t4 \ m \ \text{in } F2' (\text{butterket } i \ j) \ (\text{butterket } k \ l)) \rangle$ **for** m
have $t4 \text{inj}: x = y \ \text{if } t4 \ x = t4 \ y$ **for** $x \ y$
proof $(\text{rule } \text{ccontr})$
obtain $i \ j \ k \ l$ **where** $x = (i,j,k,l)$ **by** $(\text{meson } \text{prod-cases4})$
obtain $i' \ j' \ k' \ l'$ **where** $y = (i',j',k',l')$ **by** $(\text{meson } \text{prod-cases4})$
have $1: \text{bra } (i,k) \ *_{V} \ t4 \ x \ *_{V} \ \text{ket } (j,l) = 1$
by $(\text{auto } \text{simp}: t4\text{-def } x \ \text{tensor-op-ell2 } \text{butterfly-def } \text{cinner-ket } \text{simp } \text{flip}: \text{tensor-ell2-ket})$
assume $\langle x \neq y \rangle$
then have $2: \text{bra } (i,k) \ *_{V} \ t4 \ y \ *_{V} \ \text{ket } (j,l) = 0$
by $(\text{auto } \text{simp}: t4\text{-def } x \ y \ \text{tensor-op-ell2 } \text{butterfly-def } \text{cblinfun-apply-cblinfun-compose } \text{cinner-ket } \ \text{simp } \text{flip}: \text{tensor-ell2-ket})$
from $1 \ 2$ **that**
show *False*
by *auto*
qed
have $\langle \varphi (\text{tensor-op } (\text{butterket } i \ j) \ (\text{butterket } k \ l)) = F2' (\text{butterket } i \ j) \ (\text{butterket } k \ l) \rangle$ **for** $i \ j \ k \ l$
apply $(\text{subst } \text{asm-rl}[\text{of } \langle \text{tensor-op } (\text{butterket } i \ j) \ (\text{butterket } k \ l) = t4 \ (i,j,k,l) \rangle])$
apply $(\text{simp } \text{add}: t4\text{-def})$
by $(\text{auto } \text{simp } \text{add}: \text{injI } t4 \ \text{inj } \text{inv-f-f } \varphi\text{-def})$

have $*$: $\langle \text{range } t4 = \{ \text{tensor-op } (\text{butterket } i \ j) \ (\text{butterket } k \ l) \mid i \ j \ k \ l. \text{ True} \} \rangle$
apply $(\text{auto } \text{simp}: \text{case-prod-beta } t4\text{-def})$
using *image-iff* **by** *fastforce*

have *cblinfun-extension-exists* $(\text{range } t4) \ \varphi$
thm *cblinfun-extension-exists-finite-dim* **where** $\varphi = \varphi$
apply $(\text{rule } \text{cblinfun-extension-exists-finite-dim})$


```

  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]: ⟨cllinear (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]: ⟨cllinear (λa. ⟨x, assoc-ell2' *V a⟩)⟩ for x :: ⟨('a × 'b × 'c) ell2⟩
  by (simp add: cblinfun.add-right cinner-add-right clinearI)
  have [simp]: ⟨antilinear (λa. ⟨a, y⟩)⟩ for y :: ⟨('a × 'b × 'c) ell2⟩
  using bounded-antilinear-cinner-left bounded-antilinear-def by blast
  have [simp]: ⟨antilinear (λa. ⟨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
  ⟨λf (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-cllinear-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-

```

swap-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]*: $\langle \text{swap-ell2} * = \text{swap-ell2} \rangle$

proof (*rule adjoint-eqI[symmetric]*)

have [*simp*]: $\langle \text{cliner} (\text{cinner} (\text{swap-ell2} *_{\mathcal{V}} x)) \rangle$ **for** $x :: \langle 'a \times 'b \rangle \text{ ell2}$

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*]: $\langle \text{cliner} (\lambda a. \langle x, \text{swap-ell2} *_{\mathcal{V}} a \rangle) \rangle$ **for** $x :: \langle 'a \times 'b \rangle \text{ ell2}$

by (*simp add: cblinfun.add-right cinner-add-right clinearI*)

have [*simp*]: $\langle \text{antilinear} (\lambda a. \langle a, y \rangle) \rangle$ **for** $y :: \langle 'a \times 'b \rangle \text{ ell2}$

using *bounded-antilinear-cinner-left bounded-antilinear-def* **by** *blast*

have [*simp*]: $\langle \text{antilinear} (\lambda a. \langle \text{swap-ell2} *_{\mathcal{V}} a, y \rangle) \rangle$ **for** $y :: \langle 'b \times 'a \rangle \text{ ell2}$

by (*simp add: cblinfun.add-right cinner-add-left antilinearI*)

have $\langle \text{swap-ell2} *_{\mathcal{V}} (\text{ket } x), \text{ket } y \rangle = \langle \text{ket } x, \text{swap-ell2} *_{\mathcal{V}} \text{ket } y \rangle$ **for** $x :: \langle 'a \times 'b \rangle$ **and** y

apply (*cases x, cases y*)

by (*simp flip: tensor-ell2-ket add: swap-ell2-tensor*)

then have $\langle \text{swap-ell2} *_{\mathcal{V}} (\text{ket } x), y \rangle = \langle \text{ket } x, \text{swap-ell2} *_{\mathcal{V}} y \rangle$ **for** $x :: \langle 'a \times 'b \rangle$ **and** y

by (*rule clinear-equal-ket[THEN fun-cong, rotated 2], simp-all*)

then show $\langle \text{swap-ell2} *_{\mathcal{V}} x, y \rangle = \langle x, \text{swap-ell2} *_{\mathcal{V}} y \rangle$ **for** $x :: \langle 'a \times 'b \rangle \text{ ell2}$ **and** y

apply (*rule antilinear-equal-ket[THEN fun-cong, rotated 2]*)

by *simp-all*

qed

lemma *tensor-ell2-extensionality*:

assumes $(\bigwedge s t. a *_{\mathcal{V}} (s \otimes_s t) = b *_{\mathcal{V}} (s \otimes_s t))$

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]*: $\langle \text{assoc-ell2}' \text{ o}_{CL} \text{ assoc-ell2} = \text{id-cblinfun} \rangle$

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]*: $\langle \text{assoc-ell2} \text{ o}_{CL} \text{ assoc-ell2}' = \text{id-cblinfun} \rangle$

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*: $\langle (x + y) \otimes_o b = x \otimes_o b + y \otimes_o b \rangle$

for $x y :: \langle 'a::\text{finite ell2} \Rightarrow_{CL} 'c::\text{finite ell2} \rangle$ **and** $b :: \langle 'b::\text{finite ell2} \Rightarrow_{CL} 'd::\text{finite ell2} \rangle$

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*: $\langle b \otimes_o (x + y) = b \otimes_o x + b \otimes_o y \rangle$

for $x y :: \langle 'a::\text{finite ell2} \Rightarrow_{CL} 'c::\text{finite ell2} \rangle$ **and** $b :: \langle 'b::\text{finite ell2} \Rightarrow_{CL} 'd::\text{finite ell2} \rangle$

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*: $\langle (c *_{\mathcal{C}} x) \otimes_o b = c *_{\mathcal{C}} (x \otimes_o b) \rangle$

for $x :: \langle 'a::\text{finite ell2} \Rightarrow_{CL} 'c::\text{finite ell2} \rangle$ **and** $b :: \langle 'b::\text{finite ell2} \Rightarrow_{CL} 'd::\text{finite ell2} \rangle$

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*: $\langle b \otimes_o (c *_C x) = c *_C (b \otimes_o x) \rangle$
for $x :: \langle 'a::\text{finite ell2} \Rightarrow_{CL} 'c::\text{finite ell2} \rangle$ **and** $b :: \langle 'b::\text{finite ell2} \Rightarrow_{CL} 'd::\text{finite ell2} \rangle$
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]*: $\langle \text{clinear } (\lambda a. a \otimes_o b :: \text{finite ell2} \Rightarrow_{CL} \text{finite ell2}) \rangle$
apply (*rule clinearI*)
apply (*rule tensor-op-left-add*)
by (*rule tensor-op-scaleC-left*)

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

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

lemma *tensor-op-nonzero*:
fixes $a :: \langle 'a::\text{finite ell2} \Rightarrow_{CL} 'c::\text{finite ell2} \rangle$ **and** $b :: \langle 'b::\text{finite ell2} \Rightarrow_{CL} 'd::\text{finite ell2} \rangle$
assumes $\langle a \neq 0 \rangle$ **and** $\langle b \neq 0 \rangle$
shows $\langle a \otimes_o b \neq 0 \rangle$

proof –

from $\langle a \neq 0 \rangle$ **obtain** i **where** $i: \langle a *_V \text{ket } i \neq 0 \rangle$
by (*metis cblinfun.zero-left equal-ket*)
from $\langle b \neq 0 \rangle$ **obtain** j **where** $j: \langle b *_V \text{ket } j \neq 0 \rangle$
by (*metis cblinfun.zero-left equal-ket*)
from i j **have** $ijneq0: \langle (a *_V \text{ket } i) \otimes_s (b *_V \text{ket } j) \neq 0 \rangle$
by (*simp add: tensor-ell2-nonzero*)
have $\langle (a *_V \text{ket } i) \otimes_s (b *_V \text{ket } j) = (a \otimes_o b) *_V \text{ket } (i,j) \rangle$
by (*simp add: tensor-op-ket*)
with $ijneq0$ **show** $\langle a \otimes_o b \neq 0 \rangle$
by *force*

qed

lemma *inj-tensor-ell2-left*: $\langle \text{inj } (\lambda a::'a::\text{finite ell2}. a \otimes_s b) \rangle$ **if** $\langle b \neq 0 \rangle$ **for** $b :: \langle 'b::\text{finite ell2} \rangle$

proof (*rule injI*, *rule ccontr*)

fix x $y :: \langle 'a \text{ ell2} \rangle$
assume $eq: \langle x \otimes_s b = y \otimes_s b \rangle$
assume $neq: \langle x \neq y \rangle$
define a **where** $\langle a = x - y \rangle$
from neq a -*def* **have** $neq0: \langle a \neq 0 \rangle$
by *auto*
with $\langle b \neq 0 \rangle$ **have** $\langle a \otimes_s b \neq 0 \rangle$
by (*simp add: tensor-ell2-nonzero*)
then **have** $\langle x \otimes_s b \neq y \otimes_s b \rangle$
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*: $\langle \text{inj } (\lambda b::'b::\text{finite ell2}. a \otimes_s b) \rangle$ **if** $\langle a \neq 0 \rangle$ **for** $a :: \langle 'a::\text{finite ell2} \rangle$

proof (*rule injI*, *rule ccontr*)

fix x $y :: \langle 'b \text{ ell2} \rangle$
assume $eq: \langle a \otimes_s x = a \otimes_s y \rangle$
assume $neq: \langle x \neq y \rangle$
define b **where** $\langle b = x - y \rangle$
from neq b -*def* **have** $neq0: \langle b \neq 0 \rangle$
by *auto*
with $\langle a \neq 0 \rangle$ **have** $\langle a \otimes_s b \neq 0 \rangle$

by (simp add: tensor-ell2-nonzero)
 then have $\langle a \otimes_s x \neq a \otimes_s y \rangle$
 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: $\langle \text{inj } (\lambda a::'a::\text{finite ell2} \Rightarrow_{CL} 'c::\text{finite ell2}. a \otimes_o b) \rangle$ if $\langle b \neq 0 \rangle$ for $b :: \langle 'b::\text{finite ell2} \Rightarrow_{CL} 'd::\text{finite ell2} \rangle$

proof (rule injI, rule ccontr)
 fix $x y :: \langle 'a \text{ ell2} \Rightarrow_{CL} 'c \text{ ell2} \rangle$
 assume eq: $\langle x \otimes_o b = y \otimes_o b \rangle$
 assume neg: $\langle x \neq y \rangle$
 define a where $\langle a = x - y \rangle$
 from neg a-def have neq0: $\langle a \neq 0 \rangle$
 by auto
 with $\langle b \neq 0 \rangle$ have $\langle a \otimes_o b \neq 0 \rangle$
 by (simp add: tensor-op-nonzero)
 then have $\langle x \otimes_o b \neq y \otimes_o b \rangle$
 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: $\langle \text{inj } (\lambda b::'b::\text{finite ell2} \Rightarrow_{CL} 'c::\text{finite ell2}. a \otimes_o b) \rangle$ if $\langle a \neq 0 \rangle$ for $a :: \langle 'a::\text{finite ell2} \Rightarrow_{CL} 'd::\text{finite ell2} \rangle$

proof (rule injI, rule ccontr)
 fix $x y :: \langle 'b \text{ ell2} \Rightarrow_{CL} 'c \text{ ell2} \rangle$
 assume eq: $\langle a \otimes_o x = a \otimes_o y \rangle$
 assume neg: $\langle x \neq y \rangle$
 define b where $\langle b = x - y \rangle$
 from neg b-def have neq0: $\langle b \neq 0 \rangle$
 by auto
 with $\langle a \neq 0 \rangle$ have $\langle a \otimes_o b \neq 0 \rangle$
 by (simp add: tensor-op-nonzero)
 then have $\langle a \otimes_o x \neq a \otimes_o y \rangle$
 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 $\langle \text{tensor-ell2 } a \ b = \text{tensor-ell2 } c \ d \rangle$

assumes $\langle a \neq 0 \rangle$

shows $\langle \exists \gamma. b = \gamma * c \ d \rangle$

proof –

from $\langle a \neq 0 \rangle$ obtain i where i: $\langle \text{cinner } (\text{ket } i) \ a \neq 0 \rangle$

by (metis cinner-eq-zero-iff cinner-ket-left ell2-pointwise-ortho)

have $\langle \text{cinner } (\text{ket } i \otimes_s \text{ket } j) \ (a \otimes_s b) = \text{cinner } (\text{ket } i \otimes_s \text{ket } j) \ (c \otimes_s d) \rangle$ for j

using assms by simp

then have eq2: $\langle (\text{cinner } (\text{ket } i) \ a) * (\text{cinner } (\text{ket } j) \ b) = (\text{cinner } (\text{ket } i) \ c) * (\text{cinner } (\text{ket } j) \ d) \rangle$ for j

by (metis tensor-ell2-inner-prod)

then obtain γ where $\langle \text{cinner } (\text{ket } i) \ c = \gamma * \text{cinner } (\text{ket } i) \ a \rangle$

by (metis i eq-divide-eq)

with eq2 have $\langle (\text{cinner } (\text{ket } i) \ a) * (\text{cinner } (\text{ket } j) \ b) = (\text{cinner } (\text{ket } i) \ a) * (\gamma * \text{cinner } (\text{ket } j) \ d) \rangle$ for j

by simp

then have $\langle \text{cinner } (\text{ket } j) \ b = \text{cinner } (\text{ket } j) \ (\gamma * c \ d) \rangle$ for j

using i by force

```

then have ⟨b = γ *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 ⇒CL 'b::finite ell2⟩
  and b d :: ⟨'c::finite ell2 ⇒CL 'd::finite ell2⟩
assumes ⟨tensor-op a b = tensor-op c d⟩
assumes ⟨a ≠ 0⟩
shows ⟨∃ γ. b = γ *C d⟩
proof (cases ⟨d = 0⟩)
case False
from ⟨a ≠ 0⟩ obtain ψ where ψ: ⟨a *V ψ ≠ 0⟩
  by (metis cblinfun.zero-left cblinfun-eqI)
have ⟨(a ⊗o b) (ψ ⊗s φ) = (c ⊗o d) (ψ ⊗s φ)⟩ for φ
  using assms by simp
then have eq2: ⟨(a ψ) ⊗s (b φ) = (c ψ) ⊗s (d φ)⟩ for φ
  by (simp add: tensor-op-ell2)
then have eq2': ⟨(d φ) ⊗s (c ψ) = (b φ) ⊗s (a ψ)⟩ for φ
  by (metis swap-ell2-tensor)
from False obtain φ0 where φ0: ⟨d φ0 ≠ 0⟩
  by (metis cblinfun.zero-left cblinfun-eqI)
obtain γ where ⟨c ψ = γ *C a ψ⟩
  apply atomize-elim
  using eq2' φ0 by (rule tensor-ell2-almost-injective)
with eq2 have ⟨(a ψ) ⊗s (b φ) = (a ψ) ⊗s (γ *C d φ)⟩ for φ
  by (simp add: tensor-ell2-scaleC1 tensor-ell2-scaleC2)
then have ⟨b φ = γ *C d φ⟩ 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 = γ *C d⟩
  by (simp add: cblinfun-eqI)
then show ?thesis
  by auto
next
case True
then have ⟨c ⊗o d = 0⟩
  by (metis add-cancel-right-left tensor-op-right-add)
then have ⟨a ⊗o b = 0⟩
  using assms(1) by presburger
with ⟨a ≠ 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 ⇒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 ⇒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 $\langle \psi \neq 0 \rangle$
shows $\langle \text{bij } (\lambda x::'b::\text{finite ell2}. (\psi :: 'a::\text{CARD-1 ell2}) \otimes_s x) \rangle$

proof (rule *bijI*)

show $\langle \text{inj } (\lambda x::'b::\text{finite ell2}. (\psi :: 'a::\text{CARD-1 ell2}) \otimes_s x) \rangle$

using *assms* **by** (rule *inj-tensor-ell2-right*)

have $\langle \exists x. \psi \otimes_s x = \varphi \rangle$ **for** $\varphi :: \langle 'a * 'b \rangle \text{ ell2}$

proof (use *assms* in *transfer*)

fix $\psi :: 'a \Rightarrow \text{complex}$ **and** $\varphi :: \langle 'a * 'b \Rightarrow \text{complex} \rangle$

assume $\langle \text{has-ell2-norm } \varphi \rangle$ **and** $\langle \psi \neq (\lambda -. 0) \rangle$

define *c* **where** $\langle c = \psi \text{ undefined} \rangle$

then have $\langle \psi a = c \rangle$ **for** *a*

apply (*subst everything-the-same*[*of - undefined*])

by *simp*

with $\langle \psi \neq (\lambda -. 0) \rangle$ **have** $\langle c \neq 0 \rangle$

by *auto*

define *x* **where** $\langle x j = \varphi (\text{undefined}, j) / c \rangle$ **for** *j*

have $\langle (\lambda(i, j). \psi i * x j) = \varphi \rangle$

apply (*auto intro!*: *ext simp: x-def* $\langle \psi - = c \rangle$ $\langle c \neq 0 \rangle$)

apply (*subst* (2) *everything-the-same*[*of - undefined*])

by *simp*

then show $\langle \exists x \in \text{Collect has-ell2-norm}. (\lambda(i, j). \psi i * x j) = \varphi \rangle$

apply (rule *exI*[**where** $x=x$])

by *simp*

qed

then show $\langle \text{surj } (\lambda x::'b::\text{finite ell2}. (\psi :: 'a::\text{CARD-1 ell2}) \otimes_s x) \rangle$

by (*metis surj-def*)

qed

lemma *bij-tensor-op-one-dim-left*:

assumes $\langle a \neq 0 \rangle$

shows $\langle \text{bij } (\lambda x::'c::\text{finite ell2} \Rightarrow_{CL} 'd::\text{finite ell2}. (a :: 'a::\{\text{CARD-1,enum}\} \text{ ell2} \Rightarrow_{CL} 'b::\{\text{CARD-1,enum}\} \text{ ell2}) \otimes_o x) \rangle$

proof (rule *bijI*)

define *t* **where** $\langle t = (\lambda x::'c \text{ ell2} \Rightarrow_{CL} 'd \text{ ell2}. (a :: 'a \text{ ell2} \Rightarrow_{CL} 'b \text{ ell2}) \otimes_o x) \rangle$

define *i* **where**

$\langle i = \text{tensor-lift } (\lambda(x::'a \text{ ell2} \Rightarrow_{CL} 'b \text{ ell2}) (y::'c \text{ ell2} \Rightarrow_{CL} 'd \text{ ell2}). (\text{one-dim-iso } x / \text{one-dim-iso } a) *_C y) \rangle$

have [*simp*]: $\langle \text{clinear } i \rangle$

by (*auto intro!*: *tensor-lift-clinear simp: i-def cbilinear-def clinearI scaleC-add-left add-divide-distrib*)

have [*simp*]: $\langle \text{clinear } t \rangle$

by (*simp add: t-def*)

have $\langle i (x \otimes_o y) = (\text{one-dim-iso } x / \text{one-dim-iso } a) *_C y \rangle$ **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 $\langle t (i (x \otimes_o y)) = x \otimes_o y \rangle$ **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 $\langle t (i x) = x \rangle$ **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 $\langle \text{surj } t \rangle$

by (rule *surjI*)

show $\langle \text{inj } t \rangle$

unfolding *t-def* **using** *assms* **by** (rule *inj-tensor-right*)

qed

lemma *swap-ell2-selfinv[simp]*: $\langle \text{swap-ell2 } o_{CL} \text{ swap-ell2} = \text{id-cblinfun} \rangle$
apply (*rule tensor-ell2-extensionality*)
by *auto*

lemma *bij-tensor-op-one-dim-right*:

assumes $\langle b \neq 0 \rangle$
shows $\langle \text{bij } (\lambda x :: 'c :: \text{finite ell2} \Rightarrow_{CL} 'd :: \text{finite ell2}. x \otimes_o (b :: 'a :: \{CARD-1, \text{enum}\} \text{ ell2} \Rightarrow_{CL} 'b :: \{CARD-1, \text{enum}\} \text{ ell2})) \rangle$
(is $\langle \text{bij } ?f \rangle$ **)**

proof –

let $?sf = \langle (\lambda x. \text{swap-ell2 } o_{CL} (?f x) o_{CL} \text{swap-ell2}) \rangle$
let $?s = \langle (\lambda x. \text{swap-ell2 } o_{CL} x o_{CL} \text{swap-ell2}) \rangle$
let $?g = \langle (\lambda x :: 'c :: \text{finite ell2} \Rightarrow_{CL} 'd :: \text{finite ell2}. (b :: 'a :: \{CARD-1, \text{enum}\} \text{ ell2} \Rightarrow_{CL} 'b :: \{CARD-1, \text{enum}\} \text{ ell2}) \otimes_o x) \rangle$
have $\langle ?sf = ?g \rangle$
by (*auto intro!*: *ext tensor-ell2-extensionality simp add: swap-ell2-tensor tensor-op-ell2*)
have $\langle \text{bij } ?g \rangle$
using *assms* **by** (*rule bij-tensor-op-one-dim-left*)
have $\langle ?s o ?sf = ?f \rangle$
apply (*auto intro!*: *ext simp: cblinfun-assoc-left*)
by (*auto simp: cblinfun-assoc-right*)
also have $\langle \text{bij } ?s \rangle$
apply (*rule o-bij[where g = (\lambda 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 $\langle \text{bij } ?f \rangle$
apply (*subst* $\langle ?s o ?sf = ?f \rangle$ [*symmetric*], *subst* $\langle ?sf = ?g \rangle$)
using $\langle \text{bij } ?g \rangle \langle \text{bij } ?s \rangle$ **by** (*rule bij-comp*)

qed

lemma *overlapping-tensor*:

fixes $a23 :: \langle (a2 :: \text{finite} * a3 :: \text{finite}) \text{ ell2} \Rightarrow_{CL} (b2 :: \text{finite} * b3 :: \text{finite}) \text{ ell2} \rangle$
and $b12 :: \langle (a1 :: \text{finite} * a2) \text{ ell2} \Rightarrow_{CL} (b1 :: \text{finite} * b2) \text{ ell2} \rangle$
assumes *eq*: $\langle \text{butterfly } \psi \psi' \otimes_o a23 = \text{assoc-ell2 } o_{CL} (b12 \otimes_o \text{butterfly } \varphi \varphi') o_{CL} \text{assoc-ell2}' \rangle$
assumes $\langle \psi \neq 0 \rangle \langle \psi' \neq 0 \rangle \langle \varphi \neq 0 \rangle \langle \varphi' \neq 0 \rangle$
shows $\langle \exists c. \text{butterfly } \psi \psi' \otimes_o a23 = \text{butterfly } \psi \psi' \otimes_o c \otimes_o \text{butterfly } \varphi \varphi' \rangle$
proof –
note [*show-types*]
let $?id1 = \langle \text{id-cblinfun} :: \text{unit ell2} \Rightarrow_{CL} \text{unit ell2} \rangle$
note *id-cblinfun-eq-1[simp del]*
define *d* **where** $\langle d = \text{butterfly } \psi \psi' \otimes_o a23 \rangle$

define $\psi_n \psi_n' a23_n$ **where** $\langle \psi_n = \psi /_C \text{norm } \psi \rangle$ **and** $\langle \psi_n' = \psi' /_C \text{norm } \psi' \rangle$ **and** $\langle a23_n = \text{norm } \psi *_C \text{norm } \psi' *_C a23 \rangle$

have [*simp*]: $\langle \text{norm } \psi_n = 1 \rangle \langle \text{norm } \psi_n' = 1 \rangle$
using $\langle \psi \neq 0 \rangle \langle \psi' \neq 0 \rangle$ **by** (*auto simp: \psi_n-def \psi_n'-def norm-inverse*)
have *n1*: $\langle \text{butterfly } \psi_n \psi_n' \otimes_o a23_n = \text{butterfly } \psi \psi' \otimes_o a23 \rangle$
apply (*auto simp: \psi_n-def \psi_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 $\varphi_n \varphi_n' b12_n$ **where** $\langle \varphi_n = \varphi /_C \text{norm } \varphi \rangle$ **and** $\langle \varphi_n' = \varphi' /_C \text{norm } \varphi' \rangle$ **and** $\langle b12_n = \text{norm } \varphi *_C \text{norm } \varphi' *_C b12 \rangle$

have [*simp*]: $\langle \text{norm } \varphi_n = 1 \rangle \langle \text{norm } \varphi_n' = 1 \rangle$
using $\langle \varphi \neq 0 \rangle \langle \varphi' \neq 0 \rangle$ **by** (*auto simp: \varphi_n-def \varphi_n'-def norm-inverse*)
have *n2*: $\langle b12_n \otimes_o \text{butterfly } \varphi_n \varphi_n' = b12 \otimes_o \text{butterfly } \varphi \varphi' \rangle$
apply (*auto simp: \varphi_n-def \varphi_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 $c' :: \langle (\text{unit} * a2 * \text{unit}) \text{ ell2} \Rightarrow_{CL} (\text{unit} * b2 * \text{unit}) \text{ ell2} \rangle$

where $\langle c' = (\text{vector-to-cblinfun } \psi_n \otimes_o \text{id-cblinfun} \otimes_o \text{vector-to-cblinfun } \varphi_n) * o_{CL} d$
 $o_{CL} (\text{vector-to-cblinfun } \psi_n' \otimes_o \text{id-cblinfun} \otimes_o \text{vector-to-cblinfun } \varphi_n') \rangle$


```

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

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

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

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

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

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

```

lemma *norm-tensor-ell2*: $\langle \text{norm } (a \otimes_s b) = \text{norm } a * \text{norm } b \rangle$
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*]: $\langle \text{bounded-cbilinear } (\otimes_s) \rangle$

proof *standard*

fix $a \ a' :: 'a \ \text{ell2}$ **and** $b \ b' :: 'b \ \text{ell2}$ **and** $r :: \text{complex}$
show $\langle \text{tensor-ell2 } (a + a') \ b = \text{tensor-ell2 } a \ b + \text{tensor-ell2 } a' \ b \rangle$
by (*meson tensor-ell2-add1*)
show $\langle \text{tensor-ell2 } a \ (b + b') = \text{tensor-ell2 } a \ b + \text{tensor-ell2 } a \ b' \rangle$
by (*simp add: tensor-ell2-add2*)
show $\langle \text{tensor-ell2 } (r *_C a) \ b = r *_C \text{tensor-ell2 } a \ b \rangle$
by (*simp add: tensor-ell2-scaleC1*)
show $\langle \text{tensor-ell2 } a \ (r *_C b) = r *_C \text{tensor-ell2 } a \ b \rangle$
by (*simp add: tensor-ell2-scaleC2*)
show $\langle \exists K. \forall a \ b. \text{norm } (\text{tensor-ell2 } a \ b) \leq \text{norm } a * \text{norm } b * K \rangle$
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 (inv1 - [81] 80)*

type-synonym $'a \ \text{update} = \langle ('a \ \text{ell2}, 'a \ \text{ell2}) \ \text{cblinfun} \rangle$

lemma *preregister-mult-right*: $\langle \text{clinear } (\lambda a. a \ o_{CL} \ z) \rangle$

by (*simp add: bounded-cbilinear.add-left bounded-cbilinear-cblinfun-compose clinearI*)

lemma *preregister-mult-left*: $\langle \text{clinear } (\lambda a. z \ o_{CL} \ a) \rangle$

by (*meson cbilinear-cblinfun-compose cbilinear-def*)

definition *register* :: $\langle ('a::\text{finite update} \Rightarrow 'b::\text{finite update}) \Rightarrow \text{bool} \rangle$ **where**

register $F \longleftrightarrow$

clinear F

$\wedge F \ \text{id-cblinfun} = \text{id-cblinfun}$

$\wedge (\forall a \ b. F(a \ o_{CL} \ b) = F a \ o_{CL} \ F b)$

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

lemma *register-of-id*: $\langle \text{register } F \Longrightarrow F \ \text{id-cblinfun} = \text{id-cblinfun} \rangle$

by (*simp add: register-def*)

lemma *register-id*: $\langle \text{register } \text{id} \rangle$

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

lemma *register-preregister*: $\text{register } F \Longrightarrow \text{clinear } F$

unfolding *register-def* **by** *simp*

lemma *register-comp*: $\text{register } F \Longrightarrow \text{register } G \Longrightarrow \text{register } (G \circ F)$

unfolding *register-def*

apply *auto*

using *cllinear-compose* by *blast*

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

lemma *register-tensor-left*: $\langle \text{register } (\lambda a. \text{tensor-op } a \ \text{id-cblinfun}) \rangle$
by (*simp add: comp-tensor-op register-def tensor-op-cbilinear tensor-op-adjoint*)

lemma *register-tensor-right*: $\langle \text{register } (\lambda a. \text{tensor-op } \text{id-cblinfun } a) \rangle$
by (*simp add: comp-tensor-op register-def tensor-op-cbilinear tensor-op-adjoint*)

definition *register-pair* ::
 $\langle ('a::\text{finite update} \Rightarrow 'c::\text{finite update}) \Rightarrow ('b::\text{finite update} \Rightarrow 'c \ \text{update})$
 $\Rightarrow (('a \times 'b) \ \text{update} \Rightarrow 'c \ \text{update}) \rangle$ **where**
 $\langle \text{register-pair } F \ G = (\text{if } \text{register } F \ \wedge \ \text{register } G \ \wedge \ (\forall a \ b. F \ a \ o_{CL} \ G \ b = G \ b \ o_{CL} \ F \ a) \ \text{then}$
 $\text{tensor-lift } (\lambda a \ b. F \ a \ o_{CL} \ G \ b) \ \text{else } (\lambda \cdot. 0)) \rangle$

lemma *cbilinear-F-comp-G*[*simp*]: $\langle \text{cllinear } F \implies \text{cllinear } G \implies \text{cbilinear } (\lambda a \ b. F \ a \ o_{CL} \ G \ b) \rangle$
unfolding *cbilinear-def*
by (*auto simp add: cllinear-iff bounded-cbilinear.add-left bounded-cbilinear-cblinfun-compose bounded-cbilinear.add-right*)

lemma *register-pair-apply*:
assumes [*simp*]: $\langle \text{register } F \rangle \langle \text{register } G \rangle$
assumes $\langle \bigwedge a \ b. F \ a \ o_{CL} \ G \ b = G \ b \ o_{CL} \ F \ a \rangle$
shows $\langle (\text{register-pair } F \ G) \ (\text{tensor-op } a \ b) = F \ a \ o_{CL} \ G \ b \rangle$
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 $F :: \langle 'a::\text{finite update} \Rightarrow 'c::\text{finite update} \rangle$ **and** G
assumes [*simp*]: $\langle \text{register } F \rangle$ **and** [*simp*]: $\langle \text{register } G \rangle$
assumes $\langle \bigwedge a \ b. F \ a \ o_{CL} \ G \ b = G \ b \ o_{CL} \ F \ a \rangle$
shows $\langle \text{register } (\text{register-pair } F \ G) \rangle$
proof (*unfold register-def, intro conjI allI*)
have [*simp*]: $\langle \text{cllinear } F \rangle \langle \text{cllinear } G \rangle$
using *assms register-def* **by** *blast+*
have [*simp*]: $\langle F \ \text{id-cblinfun} = \text{id-cblinfun} \rangle \langle G \ \text{id-cblinfun} = \text{id-cblinfun} \rangle$
using *assms(1,2) register-def* **by** *blast+*
show [*simp*]: $\langle \text{cllinear } (\text{register-pair } F \ G) \rangle$
unfolding *register-pair-def*
using *assms* **apply** *auto*
apply (*rule tensor-lift-cllinear*)
by (*simp flip: assms(3)*)
show $\langle \text{register-pair } F \ G \ \text{id-cblinfun} = \text{id-cblinfun} \rangle$
apply (*simp flip: tensor-id*)
apply (*subst register-pair-apply*)
using *assms* **by** *simp-all*
have [*simp*]: $\langle \text{cllinear } (\lambda y. \text{register-pair } F \ G \ (x \ o_{CL} \ y)) \rangle$ **for** $x :: \langle ('a \times 'b) \ \text{update} \rangle$
apply (*rule cllinear-compose[unfolded o-def, where g=register-pair F G]*)
by (*simp-all add: preregister-mult-left bounded-cbilinear.add-right cllinearI*)
have [*simp*]: $\langle \text{cllinear } (\lambda y. x \ o_{CL} \ \text{register-pair } F \ G \ y) \rangle$ **for** $x :: \langle 'c \ \text{update} \rangle$
apply (*rule cllinear-compose[unfolded o-def, where f=register-pair F G]*)
by (*simp-all add: preregister-mult-left bounded-cbilinear.add-right cllinearI*)
have [*simp*]: $\langle \text{cllinear } (\lambda x. \text{register-pair } F \ G \ (x \ o_{CL} \ y)) \rangle$ **for** $y :: \langle ('a \times 'b) \ \text{update} \rangle$
apply (*rule cllinear-compose[unfolded o-def, where g=register-pair F G]*)
by (*simp-all add: bounded-cbilinear.add-left bounded-cbilinear-cblinfun-compose cllinearI*)
have [*simp*]: $\langle \text{cllinear } (\lambda x. \text{register-pair } F \ G \ x \ o_{CL} \ y) \rangle$ **for** $y :: \langle 'c \ \text{update} \rangle$
apply (*rule cllinear-compose[unfolded o-def, where f=register-pair F G]*)
by (*simp-all add: bounded-cbilinear.add-left bounded-cbilinear-cblinfun-compose cllinearI*)
have [*simp*]: $\langle F \ (x \ o_{CL} \ y) = F \ x \ o_{CL} \ F \ y \rangle$ **for** $x \ y$
by (*simp add: register-mult*)

```

have [simp]: ⟨G (x oCL y) = G x oCL G y⟩ for x y
  by (simp add: register-mult)
have [simp]: ⟨clinear (λa. (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 oCL b) = register-pair F G a oCL 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 (λb::'b::finite update. tensor-op a b)⟩
  for a :: ⟨'a::finite update⟩
proof -
  have ⟨clinear ((λb1::('a × 'b) update. (a ⊗u id-cblinfun) *u b1) o (λb. 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]: $\langle \text{clinear } (\lambda a::'a::\text{finite update. tensor-op } a \ b) \rangle$
for $b :: \langle 'b::\text{finite update} \rangle$
proof –
have $\langle \text{clinear } ((\lambda a1::('a \times 'b) \text{ update. (id-cblinfun } \otimes_u \ b) *_{\text{u}} \ a1) \ o \ (\lambda a. \text{tensor-op } a \ \text{id-cblinfun})) \rangle$
by (rule *cllinear-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 $\langle \text{register } F \rangle$
shows $\langle \text{register } (\lambda a::'a::\text{finite update. id-cblinfun } \otimes_u \ F \ a) \rangle$
using *assms apply* (rule *register-comp*[*unfolded o-def*])
by *simp*

lemma *register-tensor-id-update*[simp]:
assumes $\langle \text{register } F \rangle$
shows $\langle \text{register } (\lambda a::'a::\text{finite update. } F \ a \ \otimes_u \ \text{id-cblinfun}) \rangle$
using *assms apply* (rule *register-comp*[*unfolded o-def*])
by *simp*

11.4 Tensor product of registers

definition *register-tensor* (**infixr** \otimes_r 70) **where**
register-tensor $F \ G = \text{register-pair } (\lambda a. \text{tensor-op } (F \ a) \ \text{id-cblinfun}) \ (\lambda b. \text{tensor-op } \text{id-cblinfun } (G \ b))$

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

lemma *register-tensor-apply*[simp]:
fixes $F :: 'a::\text{finite update} \Rightarrow 'b::\text{finite update}$ **and** $G :: 'c::\text{finite update} \Rightarrow 'd::\text{finite update}$
assumes $\langle \text{register } F \rangle$ **and** $\langle \text{register } G \rangle$
shows $(F \ \otimes_r \ G) \ (a \ \otimes_u \ b) = F \ a \ \otimes_u \ G \ b$
unfolding *register-tensor-def*
apply (*subst register-pair-apply*)
unfolding *register-tensor-def*
by (*simp-all add: assms comp-tensor-op*)

definition *separating* ($-::'b::\text{finite itself}$) $A \ \longleftrightarrow$
 $(\forall F \ G :: 'a::\text{finite update} \Rightarrow 'b \ \text{update. clinear } F \ \longrightarrow \text{clinear } G \ \longrightarrow (\forall x \in A. F \ x = G \ x) \ \longrightarrow F = G)$

lemma *separating-UNIV*[simp]: $\langle \text{separating } \text{TYPE}(-) \ \text{UNIV} \rangle$
unfolding *separating-def* **by** *auto*

lemma *separating-mono*: $\langle A \subseteq B \Longrightarrow \text{separating } \text{TYPE}('a::\text{finite}) \ A \Longrightarrow \text{separating } \text{TYPE}('a) \ B \rangle$
unfolding *separating-def* **by** (*meson in-mono*)

lemma *register-eqI*: $\langle \text{separating } \text{TYPE}('b::\text{finite}) \ A \Longrightarrow \text{clinear } F \Longrightarrow \text{clinear } G \Longrightarrow (\bigwedge x. x \in A \Longrightarrow F \ x = G \ x) \Longrightarrow F = G \rangle$
unfolding *separating-def* **by** *auto*

lemma *separating-tensor*:
fixes $A :: \langle 'a::\text{finite update set} \rangle$ **and** $B :: \langle 'b::\text{finite update set} \rangle$
assumes [simp]: $\langle \text{separating } \text{TYPE}('c::\text{finite}) \ A \rangle$
assumes [simp]: $\langle \text{separating } \text{TYPE}('c) \ B \rangle$
shows $\langle \text{separating } \text{TYPE}('c) \ \{a \ \otimes_u \ b \mid a \ b. \ a \in A \ \wedge \ b \in B\} \rangle$
proof (*unfold separating-def, intro allI impI*)

```

fix F G :: ⟨('a×'b) update ⇒ 'c update⟩
assume [simp]: ⟨cllinear F⟩ ⟨cllinear G⟩
have [simp]: ⟨cllinear (λx. F (a ⊗u x))⟩ for a
  using - ⟨cllinear F⟩ apply (rule cllinear-compose[unfolded o-def])
  by simp
have [simp]: ⟨cllinear (λx. G (a ⊗u x))⟩ for a
  using - ⟨cllinear G⟩ apply (rule cllinear-compose[unfolded o-def])
  by simp
have [simp]: ⟨cllinear (λx. F (x ⊗u b))⟩ for b
  using - ⟨cllinear F⟩ apply (rule cllinear-compose[unfolded o-def])
  by simp
have [simp]: ⟨cllinear (λx. G (x ⊗u b))⟩ for b
  using - ⟨cllinear G⟩ apply (rule cllinear-compose[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::finite update set⟩ and B :: ⟨'b::finite update set⟩
assumes ⟨separating TYPE('c) 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::finite×'b::finite×'c::finite) update ⇒ 'd::finite update⟩
assumes [simp]: ⟨register F⟩ ⟨register G⟩
assumes ∧f g h. F (f ⊗u g ⊗u h) = G (f ⊗u g ⊗u h)
shows F = G
proof (rule register-eqI[where A=⟨{a⊗ub⊗uc | a b c. True}⟩])
have ⟨separating TYPE('d) {b ⊗u c | b c. True}⟩
  apply (rule separating-tensor'[where A=UNIV and B=UNIV])
  by auto
then show ⟨separating TYPE('d) {a ⊗u b ⊗u c | a b c. True}⟩
  apply (rule tac separating-tensor'[where A=UNIV and B=⟨{b⊗uc | b c. True}⟩])
  by auto
show ⟨cllinear F⟩ ⟨cllinear 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 $F G :: \langle ('a::\text{finite} \times 'b::\text{finite}) \times 'c::\text{finite} \rangle \text{update} \Rightarrow 'd::\text{finite} \text{update} \rangle$
assumes $[simp]: \langle \text{register } F \rangle \langle \text{register } G \rangle$
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** $A = \langle \{(a \otimes_u b) \otimes_u c \mid a b c. \text{True} \} \rangle$])
have $\langle \text{separating } \text{TYPE}('d) \{a \otimes_u b \mid a b. \text{True}\} \rangle$
apply (rule separating-tensor'[**where** $A = \text{UNIV}$ and $B = \text{UNIV}$])
by auto
then show $\langle \text{separating } \text{TYPE}('d) \{(a \otimes_u b) \otimes_u c \mid a b c. \text{True}\} \rangle$
apply (rule-tac separating-tensor'[**where** $B = \text{UNIV}$ and $A = \langle \{a \otimes_u b \mid a b. \text{True}\} \rangle$])
by auto
show $\langle \text{clinear } F \rangle \langle \text{clinear } G \rangle$ **by** auto
show $\langle x \in \{(a \otimes_u b) \otimes_u c \mid a b c. \text{True}\} \implies F x = G x \rangle$ **for** x
using $\text{assms}(3)$ **by** auto
qed

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

11.5 Pairs and compatibility

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

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 $x y \implies$ 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 $\langle \text{compatible } F G \rangle$
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 $\langle \text{compatible } F G \rangle$
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 $F G \implies$ register $H \implies$ compatible $(F \circ H) G$
by (simp add: compatible-def)

lemma compatible-comp-right[simp]: compatible $F G \implies$ register $H \implies$ compatible $F (G \circ H)$

by (simp add: compatible-def)

lemma compatible-comp-inner[simp]:

compatible $F G \implies$ register $H \implies$ compatible $(H \circ F) (H \circ G)$

by (smt (verit, best) comp-apply compatible-def register-comp register-mult)

lemma compatible-register1: \langle compatible $F G \implies$ register $F \rangle$

by (simp add: compatible-def)

lemma compatible-register2: \langle compatible $F G \implies$ register $G \rangle$

by (simp add: compatible-def)

lemma pair-o-tensor:

assumes compatible $A B$ and [simp]: \langle register $C \rangle$ and [simp]: \langle register $D \rangle$

shows $(A; B) \circ (C \otimes_r D) = (A \circ C; B \circ D)$

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 $F :: 'a::\text{finite update} \Rightarrow 'c::\text{finite update}$ and $G :: 'b::\text{finite update} \Rightarrow 'c::\text{finite update}$

assumes compatible $F G$

shows compatible $(\lambda a. \text{id-cblinfun} \otimes_u F a) (\lambda a. \text{id-cblinfun} \otimes_u G a)$

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

by simp

lemma compatible-tensor-id-update-right[simp]:

fixes $F :: 'a::\text{finite update} \Rightarrow 'c::\text{finite update}$ and $G :: 'b::\text{finite update} \Rightarrow 'c::\text{finite update}$

assumes compatible $F G$

shows compatible $(\lambda a. F a \otimes_u \text{id-cblinfun}) (\lambda a. G a \otimes_u \text{id-cblinfun})$

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 $(\lambda a. F a \otimes_u \text{id-cblinfun}) (\lambda a. \text{id-cblinfun} \otimes_u G a)$

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 $(\lambda a. \text{id-cblinfun} \otimes_u F a) (\lambda a. G a \otimes_u \text{id-cblinfun})$

apply (rule compatibleI)

using assms by (auto simp: comp-tensor-op)

lemma register-comp-pair:

assumes [simp]: \langle register $F \rangle$ and [simp]: \langle compatible $G H \rangle$

shows $(F \circ G; F \circ H) = F \circ (G; H)$

proof (rule tensor-extensionality)

show \langle clinear $(F \circ G; F \circ H) \rangle$ and \langle clinear $(F \circ (G; H)) \rangle$

by simp-all

have [simp]: \langle compatible $(F \circ G) (F \circ H) \rangle$

apply (rule compatible-comp-inner, simp)

by simp

then have [simp]: \langle register $(F \circ G) \rangle$ \langle register $(F \circ H) \rangle$

unfolding compatible-def by auto

from assms have [simp]: \langle register $G \rangle$ \langle register $H \rangle$

unfolding compatible-def by auto

fix $a b$

show $\langle (F \circ G; F \circ H) (a \otimes_u b) = (F \circ (G; H)) (a \otimes_u b) \rangle$

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** $\langle Fst\ a = a \otimes_u id\text{-cblinfun} \rangle$

definition *Snd* **where** $\langle Snd\ a = id\text{-cblinfun} \otimes_u a \rangle$

lemma *register-Fst[simp]*: $\langle register\ Fst \rangle$
unfolding *Fst-def* **by** (*rule register-tensor-left*)

lemma *register-Snd[simp]*: $\langle register\ Snd \rangle$
unfolding *Snd-def* **by** (*rule register-tensor-right*)

lemma *compatible-Fst-Snd[simp]*: $\langle compatible\ Fst\ Snd \rangle$
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 $\langle swap = (Snd; Fst) \rangle$

lemma *swap-apply[simp]*: $swap\ (a \otimes_u b) = (b \otimes_u a)$
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]*: $\langle register\ swap \rangle$
by (*simp add: swap-def*)

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

lemma *swap-o-swap[simp]*: $\langle swap\ o\ swap = id \rangle$
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]*: $\langle swap\ (swap\ x) = x \rangle$
by (*simp add: pointfree-idE*)

lemma *inv-swap[simp]*: $\langle inv\ swap = swap \rangle$
by (*meson inv-unique-comp swap-o-swap*)

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

lemma *register-pair-Snd*:
assumes $\langle compatible\ F\ G \rangle$
shows $\langle (F;G)\ o\ Snd = G \rangle$

```

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 (λa. (F;G) a *_u z)⟩ for z
      apply (rule clinear-compose[unfolded o-def, of ⟨(F;G)⟩])
      by simp-all
    have [simp]: ⟨clinear (λa. 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 Φ1: ⟨Φ c d σ = (F ⊗r G) σ *u (F' ⊗r G') (c ⊗u d)⟩ for c d σ
  apply (rule-tac fun-cong[of - σ])
  apply (rule tensor-extensionality)
  by auto
have ⟨Φ c d (a ⊗u b) = (F' ⊗r G') (c ⊗u d) *u (F ⊗r G) (a ⊗u b)⟩ for a b c d
  unfolding Φ-def apply (auto simp: register-pair-apply)
  by (metis assms(1) assms(2) compatible-def comp-tensor-op)
then have Φ2: ⟨Φ c d σ = (F' ⊗r G') (c ⊗u d) *u (F ⊗r G) σ⟩ for c d σ
  apply (rule-tac fun-cong[of - σ])
  apply (rule tensor-extensionality)
  by auto
from Φ1 Φ2 have ⟨(F ⊗r G) σ *u (F' ⊗r G') τ = (F' ⊗r G') τ *u (F ⊗r G) σ⟩ for τ σ
  apply (rule-tac fun-cong[of - τ])
  apply (rule tensor-extensionality)
  by auto
then show ?thesis

```

apply (*rule compatibleI[rotated -1]*)
by *auto*
qed

11.8 Associativity of the tensor product

definition *assoc* :: $\langle ('a::\text{finite} \times 'b::\text{finite}) \times 'c::\text{finite} \rangle \text{update} \Rightarrow ('a \times ('b \times 'c)) \text{update}$ **where**
 $\langle \text{assoc} = ((\text{Fst}; \text{Snd} \circ \text{Fst}); \text{Snd} \circ \text{Snd}) \rangle$

lemma *assoc-is-hom[simp]*: $\langle \text{clinear } \text{assoc} \rangle$
by (*auto simp: assoc-def*)

lemma *assoc-apply[simp]*: $\langle \text{assoc} ((a \otimes_u b) \otimes_u c) = (a \otimes_u (b \otimes_u c)) \rangle$
by (*auto simp: assoc-def register-pair-apply Fst-def Snd-def comp-tensor-op*)

definition *assoc'* :: $\langle ('a \times ('b \times 'c)) \text{update} \Rightarrow (('a::\text{finite} \times 'b::\text{finite}) \times 'c::\text{finite}) \text{update}$ **where**
 $\langle \text{assoc}' = (\text{Fst} \circ \text{Fst}; (\text{Fst} \circ \text{Snd}; \text{Snd})) \rangle$

lemma *assoc'-is-hom[simp]*: $\langle \text{clinear } \text{assoc}' \rangle$
by (*auto simp: assoc'-def*)

lemma *assoc'-apply[simp]*: $\langle \text{assoc}' (a \otimes_u (b \otimes_u c)) = ((a \otimes_u b) \otimes_u c) \rangle$
by (*auto simp: assoc'-def register-pair-apply Fst-def Snd-def comp-tensor-op*)

lemma *register-assoc[simp]*: $\langle \text{register } \text{assoc} \rangle$
unfolding *assoc-def*
by *force*

lemma *register-assoc'[simp]*: $\langle \text{register } \text{assoc}' \rangle$
unfolding *assoc'-def*
by *force*

lemma *pair-o-assoc[simp]*:
assumes [*simp*]: $\langle \text{compatible } F \ G \rangle \langle \text{compatible } G \ H \rangle \langle \text{compatible } F \ H \rangle$
shows $\langle (F; (G; H)) \circ \text{assoc} = ((F; G); H) \rangle$
proof (*rule tensor-extensionality3'*)
show $\langle \text{register} ((F; (G; H)) \circ \text{assoc}) \rangle$
by *simp*
show $\langle \text{register} ((F; G); H) \rangle$
by *simp*
show $\langle ((F; (G; H)) \circ \text{assoc}) ((f \otimes_u g) \otimes_u h) = ((F; G); H) ((f \otimes_u g) \otimes_u h) \rangle$ **for** $f \ g \ h$
by (*simp add: register-pair-apply assoc-apply cblinfun-compose-assoc*)
qed

lemma *pair-o-assoc'[simp]*:
assumes [*simp*]: $\langle \text{compatible } F \ G \rangle \langle \text{compatible } G \ H \rangle \langle \text{compatible } F \ H \rangle$
shows $\langle ((F; G); H) \circ \text{assoc}' = (F; (G; H)) \rangle$
proof (*rule tensor-extensionality3*)
show $\langle \text{register} (((F; G); H) \circ \text{assoc}') \rangle$
by *simp*
show $\langle \text{register} (F; (G; H)) \rangle$
by *simp*
show $\langle (((F; G); H) \circ \text{assoc}') (f \otimes_u g \otimes_u h) = (F; (G; H)) (f \otimes_u g \otimes_u h) \rangle$ **for** $f \ g \ h$
by (*simp add: register-pair-apply assoc'-apply cblinfun-compose-assoc*)
qed

lemma *assoc'-o-assoc[simp]*: $\langle \text{assoc}' \circ \text{assoc} = \text{id} \rangle$
apply (*rule tensor-extensionality3'*)
by *auto*

lemma *assoc'-assoc[simp]*: $\langle \text{assoc}' (\text{assoc } x) = x \rangle$
by (*simp add: pointfree-idE*)

lemma *assoc-o-assoc'*[simp]: $\langle \text{assoc } o \text{ assoc}' = \text{id} \rangle$
apply (*rule tensor-extensionality3*)
by *auto*

lemma *assoc-assoc'*[simp]: $\langle \text{assoc } (\text{assoc}' x) = x \rangle$
by (*simp add: pointfree-idE*)

lemma *inv-assoc*[simp]: $\langle \text{inv assoc} = \text{assoc}' \rangle$
using *assoc'-o-assoc assoc-o-assoc' inv-unique-comp* **by** *blast*

lemma *inv-assoc'*[simp]: $\langle \text{inv assoc}' = \text{assoc} \rangle$
by (*simp add: inv-equality*)

lemma [simp]: $\langle \text{bij assoc} \rangle$
using *assoc'-o-assoc assoc-o-assoc' o-bij* **by** *blast*

lemma [simp]: $\langle \text{bij assoc}' \rangle$
using *assoc'-o-assoc assoc-o-assoc' o-bij* **by** *blast*

11.9 Iso-registers

definition $\langle \text{iso-register } F \longleftrightarrow \text{register } F \wedge (\exists G. \text{register } G \wedge F o G = \text{id} \wedge G o F = \text{id}) \rangle$
for $F :: \langle \text{finite update} \Rightarrow \text{finite update} \rangle$

lemma *iso-registerI*:
assumes $\langle \text{register } F \rangle \langle \text{register } G \rangle \langle F o G = \text{id} \rangle \langle G o F = \text{id} \rangle$
shows $\langle \text{iso-register } F \rangle$
using *assms(1) assms(2) assms(3) assms(4) iso-register-def* **by** *blast*

lemma *iso-register-inv*: $\langle \text{iso-register } F \Longrightarrow \text{iso-register } (\text{inv } F) \rangle$
by (*metis inv-unique-comp iso-register-def*)

lemma *iso-register-inv-comp1*: $\langle \text{iso-register } F \Longrightarrow \text{inv } F o F = \text{id} \rangle$
using *inv-unique-comp iso-register-def* **by** *blast*

lemma *iso-register-inv-comp2*: $\langle \text{iso-register } F \Longrightarrow F o \text{inv } F = \text{id} \rangle$
using *inv-unique-comp iso-register-def* **by** *blast*

lemma *iso-register-id*[simp]: $\langle \text{iso-register id} \rangle$
by (*simp add: iso-register-def*)

lemma *iso-register-is-register*: $\langle \text{iso-register } F \Longrightarrow \text{register } F \rangle$
using *iso-register-def* **by** *blast*

lemma *iso-register-comp*[simp]:
assumes [simp]: $\langle \text{iso-register } F \rangle \langle \text{iso-register } G \rangle$
shows $\langle \text{iso-register } (F o G) \rangle$

proof –

from *assms* **obtain** $F' G'$ **where** [simp]: $\langle \text{register } F' \rangle \langle \text{register } G' \rangle \langle F o F' = \text{id} \rangle \langle F' o F = \text{id} \rangle$
 $\langle G o G' = \text{id} \rangle \langle G' o G = \text{id} \rangle$

by (*meson iso-register-def*)

show *?thesis*

apply (*rule iso-registerI[where G= $\langle G' o F' \rangle$]*)

apply (*auto simp: register-tensor-is-register iso-register-is-register register-tensor-distrib*)

apply (*metis $\langle F o F' = \text{id} \rangle \langle G o G' = \text{id} \rangle$ fcomp-assoc fcomp-comp id-fcomp*)

by (*metis (no-types, lifting) $\langle F o F' = \text{id} \rangle \langle F' o F = \text{id} \rangle \langle G' o G = \text{id} \rangle$ fun.map-comp inj-iff inv-unique-comp o-inv-o-cancel*)

qed

lemma *iso-register-tensor-is-iso-register*[simp]:
assumes [simp]: $\langle \text{iso-register } F \rangle \langle \text{iso-register } G \rangle$

```

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]:  $\langle \text{iso-register } I \rangle$  and  $\langle F \circ I = G \rangle$ 
  using equivalent-registers-def by blast
from assms(2) obtain J where [simp]:  $\langle \text{iso-register } J \rangle$  and  $\langle G \circ J = H \rangle$ 
  using equivalent-registers-def by blast
have  $\langle \text{register } F \rangle$ 
  by (auto simp: equivalent-registers-def)
moreover have  $\langle \text{iso-register } (I \circ J) \rangle$ 
  using  $\langle \text{iso-register } I \rangle \langle \text{iso-register } J \rangle$  iso-register-comp by blast
moreover have  $\langle F \circ (I \circ J) = H \rangle$ 
  by (simp add:  $\langle F \circ I = G \rangle \langle G \circ J = H \rangle$  o-assoc)
ultimately show ?thesis
  by (rule equivalent-registersI)
qed

```

```

lemma equivalent-registers-assoc[simp]:
  assumes [simp]:  $\langle \text{compatible } F \ G \rangle \langle \text{compatible } F \ H \rangle \langle \text{compatible } G \ H \rangle$ 
  shows  $\langle \text{equivalent-registers } (F;(G;H)) ((F;G);H) \rangle$ 
  apply (rule equivalent-registersI[where I=assoc])
  by auto

```

```

lemma equivalent-registers-pair-right:
  assumes [simp]:  $\langle \text{compatible } F \ G \rangle$ 
  assumes eq:  $\langle \text{equivalent-registers } G \ H \rangle$ 
  shows  $\langle \text{equivalent-registers } (F;G) (F;H) \rangle$ 
proof -
  from eq obtain I where [simp]:  $\langle \text{iso-register } I \rangle$  and  $\langle G \circ I = H \rangle$ 
  by (metis equivalent-registers-def)
  then have *:  $\langle (F;G) \circ (\text{id} \otimes_r I) = (F;H) \rangle$ 
  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= $\langle \text{id} \otimes_r I \rangle$ ])
  using * by (auto intro!: iso-register-tensor-is-iso-register)
qed

```

```

lemma equivalent-registers-pair-left:
  assumes [simp]:  $\langle \text{compatible } F \ G \rangle$ 
  assumes eq:  $\langle \text{equivalent-registers } F \ H \rangle$ 
  shows  $\langle \text{equivalent-registers } (F;G) (H;G) \rangle$ 
proof -
  from eq obtain I where [simp]:  $\langle \text{iso-register } I \rangle$  and  $\langle F \circ I = H \rangle$ 
  by (metis equivalent-registers-def)
  then have *:  $\langle (F;G) \circ (I \otimes_r \text{id}) = (H;G) \rangle$ 
  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= $\langle I \otimes_r \text{id} \rangle$ ])
  using * by (auto intro!: iso-register-tensor-is-iso-register)
qed

```

```

lemma equivalent-registers-comp:
  assumes  $\langle \text{register } H \rangle$ 
  assumes  $\langle \text{equivalent-registers } F \ G \rangle$ 
  shows  $\langle \text{equivalent-registers } (H \circ F) (H \circ G) \rangle$ 
  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) =  $\langle$ 
  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 or else 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** ⊗_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

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* ($\lambda b1\ b2.$ *case* (*b1*, *b2*) *of* (*0*, *0*) \Rightarrow *order.Eq* | (*0*, *1*) \Rightarrow *order.Lt* | (*1*, *0*) \Rightarrow *order.Gt* | (*1*, *1*) \Rightarrow *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* :: \langle (*bit*, *bit*) *matrix* \rangle **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* :: \langle (*bit*, *bit*) *matrix* \rangle **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* :: \langle (*bit***bit*, *bit***bit*) *matrix* \rangle **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]*: \langle *CNOT* *_V *ket* (*i*, *j*) = *ket* (*i*, *j*+*i*) \rangle

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* :: $\langle (\text{bit} \times \text{bit}, \text{bit} \times \text{bit}) \text{ matrix} \rangle$ **where**
 [code del]: $\langle \text{Uswap} = \text{cblinfun-of-mat matrix-Uswap} \rangle$

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]: $\langle \text{Uswap} *_V s \otimes_s t = t \otimes_s s \rangle$
apply (*rule clinear-equal-ket*[**where** $f = \langle \lambda s. \text{Uswap} *_V s \otimes_s t \rangle$, *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** $f = \langle \lambda t. \text{Uswap} *_V - \otimes_s t \rangle$, *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** \sqcap_1 70)

no-notation *Group.mult* (**infixl** \otimes_1 70)

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

unbundle *register-notation*

unbundle *cblinfun-notation*

lemma *zero-not-register*[simp]: $\langle \sim \text{register } (\lambda. 0) \rangle$
unfolding *register-def* **by** *simp*

lemma *register-pair-is-register-converse*:
 $\langle \text{register } (F;G) \implies \text{register } F \rangle \langle \text{register } (F;G) \implies \text{register } G \rangle$
using [[*simproc del: Laws-Quantum.compatibility-warn*]]
apply (*cases* $\langle \text{register } F \rangle$)
apply (*auto simp: register-pair-def*)[2]
apply (*cases* $\langle \text{register } G \rangle$)
by (*auto simp: register-pair-def*)[2]

lemma *register-id'*[simp]: $\langle \text{register } (\lambda x. x) \rangle$
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 $(R a *_S \top) \sqcap (S b *_S \top) = ((R a \text{ o}_{CL} S b) *_S \top)$
proof (*rule antisym*)
have $((R a \text{ o}_{CL} S b) *_S \top) \leq (S b *_S \top)$
apply (*subst swap-registers[OF assms(1)]*)
by (*simp add: cblinfun-compose-image cblinfun-image-mono*)
moreover have $((R a \text{ o}_{CL} S b) *_S \top) \leq (R a *_S \top)$
by (*simp add: cblinfun-compose-image cblinfun-image-mono*)
ultimately show $\langle ((R a \text{ o}_{CL} S b) *_S \top) \leq (R a *_S \top) \sqcap (S b *_S \top) \rangle$
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 $\langle (R a *_S \top) \sqcap (S b *_S \top) \leq (R a \text{ o}_{CL} S b) *_S \top \rangle$
proof (*unfold less-eq-ccsubspace.rep-eq, rule*)
fix ψ
assume *asm*: $\langle \psi \in \text{space-as-set } ((R a *_S \top) \sqcap (S b *_S \top)) \rangle$
then have $\langle \psi \in \text{space-as-set } (R a *_S \top) \rangle$
by *auto*
then have *R*: $\langle R a *_V \psi = \psi \rangle$
using $\langle \text{is-Proj } (R a) \rangle$ *cblinfun-fixes-range is-Proj-algebraic by blast*
from *asm* **have** $\langle \psi \in \text{space-as-set } (S b *_S \top) \rangle$
by *auto*
then have *S*: $\langle S b *_V \psi = \psi \rangle$
using $\langle \text{is-Proj } (S b) \rangle$ *cblinfun-fixes-range is-Proj-algebraic by blast*
from *R S* **have** $\langle \psi = (R a \text{ o}_{CL} S b) *_V \psi \rangle$
by (*simp add: cblinfun-apply-cblinfun-compose*)
also have $\langle \dots \in \text{space-as-set } ((R a \text{ o}_{CL} S b) *_S \top) \rangle$
apply *simp by (metis R S calculation cblinfun-apply-in-image)*
finally show $\langle \psi \in \text{space-as-set } ((R a \text{ o}_{CL} S b) *_S \top) \rangle$
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 [*simp proc 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*: $\langle \text{unitary } a \implies \text{register } (\text{sandwich } a) \rangle$
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 $a :: \langle 'a::\text{finite ell2} \Rightarrow_{CL} 'a \text{ ell2} \rangle$ and $b :: \langle 'b::\text{finite ell2} \Rightarrow_{CL} 'b \text{ ell2} \rangle$

assumes $\langle \text{unitary } a \rangle \langle \text{unitary } b \rangle$

shows $\text{sandwich } (a \otimes_o b) = \text{sandwich } a \otimes_r \text{sandwich } b$

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 $a :: \langle 'a::\text{finite ell2} \Rightarrow_{CL} 'a \text{ ell2} \rangle$

assumes $\text{unitary } a$

shows $\text{sandwich } a \otimes_r \text{id} = \text{sandwich } (a \otimes_o \text{id-cblinfun})$

by (simp add: unitary-sandwich-register assms sandwich-tensor sandwich-id)

lemma register-sandwich: $\langle \text{register } F \implies F (\text{sandwich } a b) = \text{sandwich } (F a) (F b) \rangle$

by (smt (verit, del-insts) register-def sandwich-def)

lemma assoc-ell2-sandwich: $\langle \text{assoc} = \text{sandwich } \text{assoc-ell2} \rangle$

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: $\langle \text{assoc}' = \text{sandwich } \text{assoc-ell2}' \rangle$

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: $\text{swap} = \text{sandwich } U\text{swap}$

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 $a :: 'a::\text{finite ell2} \Rightarrow_{CL} 'b::\text{finite ell2}$

assumes $\text{unitary } a$

shows $\text{id} \otimes_r \text{sandwich } a = \text{sandwich } (\text{id-cblinfun} \otimes_o a)$

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 $R (\text{selfbutter } \psi) \circ_{CL} S (\text{selfbutter } \varphi) = (R; S) (\text{selfbutter } (\psi \otimes_s \varphi))$

apply (subst register-pair-apply[symmetric, where F=R and G=S])

using assms by auto

lemma register-mult':

assumes $\langle \text{register } F \rangle$

shows $\langle F a *_V F b *_V c = F (a \circ_{CL} b) *_V c \rangle$

by (simp add: assms lift-cblinfun-comp(4) register-mult)

lemma register-scaleC:

assumes $\langle \text{register } F \rangle$ **shows** $\langle F (c *_C a) = c *_C F a \rangle$
by (*simp add: assms complex-vector.linear-scale*)

lemma *register-bounded-clinear*: $\langle \text{register } F \implies \text{bounded-clinear } F \rangle$
using *bounded-clinear-finite-dim register-def* **by** *blast*

lemma *register-adjoint*: $F (a^*) = (F a)^*$ **if** $\langle \text{register } F \rangle$
using *register-def* **that** **by** *blast*

end

14 Very simple Quantum Hoare logic

theory *QHoare*
imports *Quantum-Extra*
begin

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

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

definition *apply* $U R = R U$ **for** $R :: \langle 'a \text{ update} \Rightarrow 'mem \text{ update} \rangle$
definition *ifthen* $R x = R (\text{butterket } x x)$ **for** $R :: \langle 'a \text{ update} \Rightarrow 'mem \text{ update} \rangle$
definition *program* $S = \text{fold } (o_{CL}) S \text{ id-cblinfun}$ **for** $S :: \langle 'mem \text{ update list} \rangle$

definition *hoare* :: $\langle 'mem \text{ ell2 ccspace} \Rightarrow ('mem \text{ ell2} \Rightarrow_{CL} 'mem \text{ ell2}) \text{ list} \Rightarrow 'mem \text{ ell2 ccspace} \Rightarrow \text{bool} \rangle$
where
hoare $C p D \longleftrightarrow (\forall \psi \in \text{space-as-set } C. \text{program } p *_V \psi \in \text{space-as-set } D)$ **for** $C p D$

definition *EQ* :: $\langle 'a \text{ update} \Rightarrow 'mem \text{ update} \rangle \Rightarrow 'a \text{ ell2} \Rightarrow 'mem \text{ ell2 ccspace}$ (**infix** $=_q$ 75) **where**
 $EQ R \psi = R (\text{selfbutter } \psi) *_S \top$

lemma *program-skip*[*simp*]: $\text{program } [] = \text{id-cblinfun}$
by (*simp add: qhoare.program-def*)

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

lemma *hoare-seq*[*trans*]: $\text{hoare } C p1 D \implies \text{hoare } D p2 E \implies \text{hoare } C (p1 @ p2) E$
by (*auto simp: program-seq hoare-def*)

lemma *hoare-weaken-left*[*trans*]: $\langle A \leq B \implies \text{hoare } B p C \implies \text{hoare } A p C \rangle$
unfolding *hoare-def*
by (*meson in-mono less-eq-ccspace.rep-eq*)

lemma *hoare-weaken-right*[*trans*]: $\langle \text{hoare } A p B \implies B \leq C \implies \text{hoare } A p C \rangle$
unfolding *hoare-def*
by (*meson in-mono less-eq-ccspace.rep-eq*)

lemma *hoare-skip*: $C \leq D \implies \text{hoare } C [] D$
by (*auto simp: program-def hoare-def in-mono less-eq-ccspace.rep-eq*)

lemma *hoare-apply*:
assumes $R U *_S \text{pre} \leq \text{post}$
shows $\text{hoare } \text{pre} [\text{apply } U R] \text{post}$
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-ccspace.rep-eq subsetD*)

```

lemma hoare-iffthen:
  fixes R :: ⟨'a update ⇒ 'mem update⟩
  assumes R (selfbutter (ket x)) *s pre ≤ post
  shows hoare pre [iffthen R x] post
  using assms
  apply (auto simp: hoare-def program-def iffthen-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 ⇒ nat ⇒ (nat × nat) ⇒ nat
  where tensor-pack X Y = (λ(x, y). x * Y + y)

```

```

definition tensor-unpack :: nat ⇒ nat ⇒ nat ⇒ (nat × 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 ⟷ 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 ⟹ fst (tensor-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 ⟹ snd (tensor-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: ⟨fst (tensor-unpack A B (fst (tensor-unpack (A * B) C i)))
  = fst (tensor-unpack A (B * C) i)⟩
  unfolding tensor-unpack-def apply auto
  by (metis div-mult2-eq mult.commute)

```

```

lemma tensor-unpack-sndsnd: ⟨snd (tensor-unpack B C (snd (tensor-unpack A (B * C) i)))
  = snd (tensor-unpack (A * B) C i)⟩
  unfolding tensor-unpack-def apply auto
  by (meson dvd-triv-right mod-mod-cancel)

```

```

lemma tensor-unpack-fstsnd: ⟨fst (tensor-unpack B C (snd (tensor-unpack A (B * C) i)))
  = snd (tensor-unpack A B (fst (tensor-unpack (A * B) C i)))⟩
  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 ψ φ = (let d1 = dim-vec ψ in let d2 = dim-vec φ in
  vec (d1*d2) (λi. let (i1,i2) = tensor-unpack d1 d2 i in (vec-index ψ i1) * (vec-index φ i2)))

```

```

lemma tensor-state-jnf-dim[simp]: ⟨dim-vec (tensor-state-jnf ψ φ) = dim-vec ψ * dim-vec φ⟩
  unfolding tensor-state-jnf-def Let-def by simp

```

```

lemma enum-prod-nth-tensor-unpack:
  assumes ⟨i < CARD('a) * CARD('b)⟩
  shows (Enum.enum ! i :: 'a::enum × '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  $\psi :: \langle 'a::\text{enum ell2} \rangle$  and  $\varphi :: \langle 'b::\text{enum ell2} \rangle$ 
  assumes [simp]:  $\langle i < \text{CARD}('a) * \text{CARD}('b) \rangle$ 
  shows  $\langle \text{vec-of-basis-enum } (\psi \otimes_s \varphi) \$ i = (\text{let } (i1,i2) = \text{tensor-unpack } \text{CARD}('a) \text{ CARD}('b) \text{ i in}$ 
     $\text{vec-of-basis-enum } \psi \$ i1 * \text{vec-of-basis-enum } \varphi \$ i2) \rangle$ 
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  $\langle \text{vec-of-basis-enum } (\psi \otimes_s \varphi) \$ i = \text{Rep-ell2 } (\psi \otimes_s \varphi) (\text{enum-class.enum ! } i) \rangle$ 
    by (simp add: vec-of-basis-enum-ell2-component)
  also have  $\langle \dots = \text{Rep-ell2 } \psi (\text{Enum.enum!}i1) * \text{Rep-ell2 } \varphi (\text{Enum.enum!}i2) \rangle$ 
    apply (transfer fixing: i i1 i2)
    by (simp add: enum-prod-nth-tensor-unpack case-prod-beta i1-def i2-def)
  also have  $\langle \dots = \text{vec-of-basis-enum } \psi \$ i1 * \text{vec-of-basis-enum } \varphi \$ i2 \rangle$ 
    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  $\psi :: \langle 'a::\text{enum ell2} \rangle$  and  $\varphi :: \langle 'b::\text{enum ell2} \rangle$ 
  shows  $\langle \text{vec-of-basis-enum } (\psi \otimes_s \varphi) = \text{tensor-state-jnf } (\text{vec-of-basis-enum } \psi) (\text{vec-of-basis-enum } \varphi) \rangle$ 
  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 ::  $\langle 'a::\text{enum ell2} \Rightarrow_{CL} 'b::\text{enum ell2} \rangle$  and b ::  $\langle 'c::\text{enum ell2} \Rightarrow_{CL} 'd::\text{enum ell2} \rangle$ 
  assumes [simp]:  $\langle i < \text{CARD}('b) * \text{CARD}('d) \rangle$ 
  assumes [simp]:  $\langle j < \text{CARD}('a) * \text{CARD}('c) \rangle$ 
  shows  $\langle \text{mat-of-cblinfun } (\text{tensor-op } a \ b) \$\$ (i,j) =$ 
    (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)) \rangle
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  $\langle \text{mat-of-cblinfun } (\text{tensor-op } a \ b) \$\$ (i,j)$ 
    =  $\text{Rep-ell2 } (\text{tensor-op } a \ b *_{\vee} \text{ket } (\text{Enum.enum!}j)) (\text{Enum.enum ! } i) \rangle$ 
    by (simp add: mat-of-cblinfun-ell2-component)
  also have  $\langle \dots = \text{Rep-ell2 } ((a *_{\vee} \text{ket } (\text{Enum.enum!}j1)) \otimes_s (b *_{\vee} \text{ket } (\text{Enum.enum!}j2))) (\text{Enum.enum!}i) \rangle$ 
    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  $\langle \dots = \text{vec-of-basis-enum } ((a *_{\vee} \text{ket } (\text{Enum.enum!}j1)) \otimes_s b *_{\vee} \text{ket } (\text{Enum.enum!}j2)) \$ i \rangle$ 

```


by (simp add: vec-of-basis-enum-ell2-component)
 also have $\langle \dots = \text{vec-of-basis-enum } (a *_{\mathbb{V}} \text{ket } (\text{enum-class.enum } ! j1)) \$ i1 * \text{vec-of-basis-enum } (b *_{\mathbb{V}} \text{ket } (\text{enum-class.enum } ! j2)) \$ i2 \rangle$
 by (simp add: case-prod-beta vec-of-basis-enum-tensor-state-index i1-def[symmetric] i2-def[symmetric])
 also have $\langle \dots = \text{Rep-ell2 } (a *_{\mathbb{V}} \text{ket } (\text{enum-class.enum } ! j1)) (\text{enum-class.enum } ! i1) * \text{Rep-ell2 } (b *_{\mathbb{V}} \text{ket } (\text{enum-class.enum } ! j2)) (\text{enum-class.enum } ! i2) \rangle$
 by (simp add: vec-of-basis-enum-ell2-component)
 also have $\langle \dots = \text{mat-of-cblinfun } a \$\$ (i1, j1) * \text{mat-of-cblinfun } b \$\$ (i2, j2) \rangle$
 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 $r1 = \text{dim-row } A$ in
 let $c1 = \text{dim-col } A$ in
 let $r2 = \text{dim-row } B$ in
 let $c2 = \text{dim-col } B$ in
 mat $(r1*r2)$ $(c1*c2)$
 $(\lambda(i,j). \text{let } (i1,i2) = \text{tensor-unpack } r1 r2 i \text{ in}$
 $\text{let } (j1,j2) = \text{tensor-unpack } c1 c2 j \text{ in}$
 $(A \$\$ (i1,j1)) * (B \$\$ (i2,j2))))$

lemma *tensor-op-jnf-dim*[simp]:

$\langle \text{dim-row } (\text{tensor-op-jnf } a b) = \text{dim-row } a * \text{dim-row } b \rangle$
 $\langle \text{dim-col } (\text{tensor-op-jnf } a b) = \text{dim-col } a * \text{dim-col } b \rangle$
 unfolding *tensor-op-jnf-def* Let-def by simp-all

lemma *mat-of-cblinfun-tensor-op*:

fixes $a :: \langle 'a::\text{enum } \text{ell2} \Rightarrow_{CL} 'b::\text{enum } \text{ell2} \rangle$ and $b :: \langle 'c::\text{enum } \text{ell2} \Rightarrow_{CL} 'd::\text{enum } \text{ell2} \rangle$
 shows $\langle \text{mat-of-cblinfun } (\text{tensor-op } a b) = \text{tensor-op-jnf } (\text{mat-of-cblinfun } a) (\text{mat-of-cblinfun } b) \rangle$
 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]:

$\langle \text{mat-of-cblinfun } (\text{assoc-ell2}' :: (('a::\text{enum} \times ('b::\text{enum} \times 'c::\text{enum})) \text{ell2} \Rightarrow_{CL} -)) = \text{one-mat } (\text{CARD}'a) * \text{CARD}'b * \text{CARD}'c \rangle$
 (is *mat-of-cblinfun* ?assoc = -)

proof (rule *mat-eq-iff*[THEN *iffD2*], intro *conjI* *allI* *impI*)

show $\langle \text{dim-row } (\text{mat-of-cblinfun } ?\text{assoc}) = \text{dim-row } (1_m (\text{CARD}'a) * \text{CARD}'b * \text{CARD}'c)) \rangle$
 by (simp)

show $\langle \text{dim-col } (\text{mat-of-cblinfun } ?\text{assoc}) = \text{dim-col } (1_m (\text{CARD}'a) * \text{CARD}'b * \text{CARD}'c)) \rangle$
 by (simp)

fix $i j$

let $?i = \text{Enum.enum } ! i :: (('a \times 'b) \times 'c)$ and $?j = \text{Enum.enum } ! j :: ('a \times ('b \times 'c))$

assume $\langle i < \text{dim-row } (1_m (\text{CARD}'a) * \text{CARD}'b * \text{CARD}'c)) \rangle$
 then have iB [simp]: $\langle i < \text{CARD}'a * \text{CARD}'b * \text{CARD}'c \rangle$ by simp
 then have iB' [simp]: $\langle i < \text{CARD}'a * (\text{CARD}'b * \text{CARD}'c) \rangle$ by *linarith*
 assume $\langle j < \text{dim-col } (1_m (\text{CARD}'a) * \text{CARD}'b * \text{CARD}'c)) \rangle$
 then have jB [simp]: $\langle j < \text{CARD}'a * \text{CARD}'b * \text{CARD}'c \rangle$ by simp
 then have jB' [simp]: $\langle j < \text{CARD}'a * (\text{CARD}'b * \text{CARD}'c) \rangle$ by *linarith*

define $i1 i23 i2 i3$

where $i1 = \text{fst } (\text{tensor-unpack } \text{CARD}'a) (\text{CARD}'b * \text{CARD}'c) i$
 and $i23 = \text{snd } (\text{tensor-unpack } \text{CARD}'a) (\text{CARD}'b * \text{CARD}'c) i$

```

    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 oCL ?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-Numerals
  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 Φ1 ≡ Φ o Fst
abbreviation Φ2 ≡ Φ o Snd
abbreviation XΦ2 ≡ (X;Φ2)
abbreviation XΦ1 ≡ (X;Φ1)
abbreviation XΦ ≡ (X;Φ)
abbreviation XAB ≡ ((X;A); B)
abbreviation AB ≡ (A;B)
abbreviation Φ2AB ≡ ((Φ o Snd; A); B)

definition teleport a b = [
  apply CNOT XΦ1,
  apply hadamard X,
  ifthen Φ1 a,
  ifthen X b,
  apply (if a=1 then pauliX else id-cblinfun) Φ2,

```

```

  apply (if b=1 then pauliZ else id-cblinfun)  $\Phi$ 2
]

```

lemma Φ - $X\Phi$: $\langle \Phi a = X\Phi (id-cblinfun \otimes_o a) \rangle$

by (auto simp: register-pair-apply)

lemma $X\Phi$ 1- $X\Phi$: $\langle X\Phi 1 a = X\Phi (assoc (a \otimes_o id-cblinfun)) \rangle$

apply (subst pair-o-assoc[unfolded o-def, of X Φ 1 Φ 2, simplified, THEN fun-cong])

by (auto simp: register-pair-apply)

lemma $X\Phi$ 2- $X\Phi$: $\langle X\Phi 2 a = X\Phi ((id \otimes_r swap) (assoc (a \otimes_o id-cblinfun))) \rangle$

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 Φ 2- $X\Phi$: $\langle \Phi 2 a = X\Phi (id-cblinfun \otimes_o (id-cblinfun \otimes_o a)) \rangle$

by (auto simp: Snd-def register-pair-apply)

lemma X - $X\Phi$: $\langle X a = X\Phi (a \otimes_o id-cblinfun) \rangle$

by (auto simp: register-pair-apply)

lemma Φ 1- $X\Phi$: $\langle \Phi 1 a = X\Phi (id-cblinfun \otimes_o (a \otimes_o id-cblinfun)) \rangle$

by (auto simp: Fst-def register-pair-apply)

lemmas to- $X\Phi$ = Φ - $X\Phi$ $X\Phi$ 1- $X\Phi$ $X\Phi$ 2- $X\Phi$ Φ 2- $X\Phi$ X - $X\Phi$ Φ 1- $X\Phi$

lemma X - $X\Phi$ 1: $\langle X a = X\Phi 1 (a \otimes_o id-cblinfun) \rangle$

by (auto simp: register-pair-apply)

lemmas to- $X\Phi$ 1 = X - $X\Phi$ 1

lemma XAB -to- $X\Phi$ 2- AB : $\langle XAB a = (X\Phi 2; AB) ((swap \otimes_r id) (assoc' (id-cblinfun \otimes_o assoc a))) \rangle$

by (simp add: pair-o-tensor[unfolded o-def, THEN fun-cong] register-pair-apply

pair-o-swap[unfolded o-def, THEN fun-cong]

pair-o-assoc'[unfolded o-def, THEN fun-cong]

pair-o-assoc[unfolded o-def, THEN fun-cong])

lemma $X\Phi$ 2-to- $X\Phi$ 2- AB : $\langle X\Phi 2 a = (X\Phi 2; AB) (a \otimes_o id-cblinfun) \rangle$

by (simp add: register-pair-apply)

schematic-goal Φ 2 AB -to- $X\Phi$ 2- AB : Φ 2 AB $a = (X\Phi 2; AB) ?b$

apply (subst pair-o-assoc'[unfolded o-def, THEN fun-cong])

apply simp-all[β]

apply (subst register-pair-apply[where $a=id-cblinfun$])

apply simp-all[β]

apply (subst pair-o-assoc[unfolded o-def, THEN fun-cong])

apply simp-all[β]

by simp

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

lemma teleport:

assumes [simp]: norm $\psi = 1$

shows hoare $(XAB =_q \psi \sqcap \Phi =_q \beta 00)$ (teleport a b) $(\Phi 2AB =_q \psi)$

proof –

define XZ :: \langle bit update \rangle **where** $XZ = (if a=1 then (if b=1 then pauliZ o_{CL} pauliX else pauliX) else (if b=1 then pauliZ else id-cblinfun))$

define pre **where** $pre = XAB =_q \psi$

define $O1$ **where** $O1 = \Phi (selfbutter \beta 00)$

have $\langle (XAB =_q \psi \sqcap \Phi =_q \beta 00) = O1 *_S pre \rangle$

unfolding pre -def $O1$ -def EQ-def

apply (subst compatible-proj-intersect[where $R=XAB$ and $S=\Phi$])

apply (simp-all add: butterfly-is-Proj)

apply (subst swap-registers[where $R=XAB$ and $S=\Phi$])

by (simp-all add: cblinfun-assoc-left(2))

also
define $O2$ where $O2 = X\Phi1\ CNOT\ o_{CL}\ O1$
have $\langle hoare\ (O1\ *_S\ pre)\ [apply\ CNOT\ X\Phi1]\ (O2\ *_S\ pre)\rangle$
apply (rule hoare-apply) **by** (simp add: O2-def cblinfun-assoc-left(2))

also
define $O3$ where $\langle O3 = X\ hadamard\ o_{CL}\ O2\rangle$
have $\langle hoare\ (O2\ *_S\ pre)\ [apply\ hadamard\ X]\ (O3\ *_S\ pre)\rangle$
apply (rule hoare-apply) **by** (simp add: O3-def cblinfun-assoc-left(2))

also
define $O4$ where $\langle O4 = \Phi1\ (selfbutterket\ a)\ o_{CL}\ O3\rangle$
have $\langle hoare\ (O3\ *_S\ pre)\ [ifthen\ \Phi1\ a]\ (O4\ *_S\ pre)\rangle$
apply (rule hoare-ifthen) **by** (simp add: O4-def cblinfun-assoc-left(2))

also
define $O5$ where $\langle O5 = X\ (selfbutterket\ b)\ o_{CL}\ O4\rangle$
have $\langle hoare\ (O4\ *_S\ pre)\ [ifthen\ X\ b]\ (O5\ *_S\ pre)\rangle$
apply (rule hoare-ifthen) **by** (simp add: O5-def cblinfun-assoc-left(2))

also
define $O6$ where $\langle O6 = \Phi2\ (if\ a=1\ then\ pauliX\ else\ id-cblinfun)\ o_{CL}\ O5\rangle$
have $\langle hoare\ (O5\ *_S\ pre)\ [apply\ (if\ a=1\ then\ pauliX\ else\ id-cblinfun)\ (\Phi\ o\ Snd)]\ (O6\ *_S\ pre)\rangle$
apply (rule hoare-apply) **by** (auto simp add: O6-def cblinfun-assoc-left(2))

also
define $O7$ where $\langle O7 = \Phi2\ (if\ b = 1\ then\ pauliZ\ else\ id-cblinfun)\ o_{CL}\ O6\rangle$
have $O7$: $\langle O7 = \Phi2\ XZ\ o_{CL}\ O5\rangle$
by (auto simp add: O6-def O7-def XZ-def register-mult lift-cblinfun-comp[OF register-mult])
have $\langle hoare\ (O6\ *_S\ pre)\ [apply\ (if\ b=1\ then\ pauliZ\ else\ id-cblinfun)\ (\Phi\ o\ Snd)]\ (O7\ *_S\ pre)\rangle$
apply (rule hoare-apply)
by (auto simp add: O7-def cblinfun-assoc-left(2))

finally have $hoare$: $\langle hoare\ (XAB =_q\ \psi\ \sqcap\ \Phi =_q\ \beta00)\ (teleport\ a\ b)\ (O7\ *_S\ pre)\rangle$
by (auto simp add: teleport-def comp-def)

have $O5'$: $O5 = (1/2) *_C\ \Phi2\ (XZ*)\ o_{CL}\ X\Phi2\ Uswap\ o_{CL}\ \Phi\ (butterfly\ (ket\ a\ \otimes_s\ ket\ b)\ \beta00)$
unfolding $O7\ O5\text{-def}\ O4\text{-def}\ O3\text{-def}\ O2\text{-def}\ O1\text{-def}$
apply (simp split del: if-split only: to-X Φ register-mult[of X Φ])
apply (simp split del: if-split add: register-mult[of X Φ])
 flip: complex-vector.linear-scale
 del: comp-apply)
apply (rule arg-cong[of - - X Φ])
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 $O7'$: $O7 = (1/2) *_C\ X\Phi2\ Uswap\ o_{CL}\ \Phi\ (butterfly\ (ket\ a\ \otimes_s\ ket\ b)\ \beta00)$
unfolding $O7\ O5'$
by (simp add: cblinfun-compose-assoc[symmetric] register-mult[of $\Phi2$] 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 [[simp 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* (*inv1* - [81] 80)

no-notation *Lattice.join* (**infixl** \sqcup_1 65)

typedef (*'a::finite, 'b::finite*) *complement-domain* = $\langle \{.. < \text{if } \text{CARD}('b) \text{ div } \text{CARD}('a) \neq 0 \text{ then } \text{CARD}('b) \text{ div } \text{CARD}('a) \text{ else } 1\} \rangle$

by *auto*

instance *complement-domain* :: (*finite, finite*) *finite*

proof *intro-classes*

have $\langle \text{inj } \text{Rep-complement-domain} \rangle$

by (*simp add: Rep-complement-domain-inject inj-on-def*)

moreover have $\langle \text{finite } (\text{range } \text{Rep-complement-domain}) \rangle$

by (*metis finite-lessThan type-definition.Rep-range type-definition-complement-domain*)

ultimately show $\langle \text{finite } (\text{UNIV} :: ('a, 'b) \text{ complement-domain set}) \rangle$

using *finite-image-iff* **by** *blast*

qed

lemma *CARD-complement-domain*:

assumes $\langle \text{CARD}('b::\text{finite}) = n * \text{CARD}('a::\text{finite}) \rangle$

shows $\langle \text{CARD}((('a, 'b) \text{ complement-domain}) = n) \rangle$

proof –

from *assms* **have** $\langle n > 0 \rangle$

by (*metis zero-less-card-finite zero-less-mult-pos2*)

have $\ast: \langle \text{inj } \text{Rep-complement-domain} \rangle$

by (*simp add: Rep-complement-domain-inject inj-on-def*)

moreover have $\langle \text{card } (\text{range } (\text{Rep-complement-domain} :: ('a, 'b) \text{ complement-domain} \Rightarrow -)) = n \rangle$

apply (*subst type-definition.Rep-range[OF type-definition-complement-domain]*)

using *assms* $\langle n > 0 \rangle$ **by** *simp*

ultimately show *?thesis*

by (*metis card-image*)

qed

lemma *register-decomposition*:

fixes $\Phi :: \langle 'a::\text{finite update} \Rightarrow 'b::\text{finite update} \rangle$

assumes [*simp*]: $\langle \text{register } \Phi \rangle$

shows $\langle \exists U :: \langle 'a \times ('a, 'b) \text{ complement-domain} \rangle \text{ ell2} \Rightarrow_{CL} 'b \text{ ell2. unitary } U \wedge (\forall \vartheta. \Phi \vartheta = \text{sandwich } U (\vartheta \otimes_{\circ} \text{id-cblinfun})) \rangle$

– Proof based on [Daw21]

proof –

note [*simproc del: compatibility-warn*]

fix $\xi 0 :: 'a$

have [*simp*]: $\langle \text{clinear } \Phi \rangle$

by *simp*

define *P* **where** $\langle P \ i = \text{Proj } (\text{ccspan } \{\text{ket } i\}) \rangle$ **for** $i :: 'a$

have *P-butter*: $\langle P \ i = \text{selfbutterket } i \rangle$ **for** i

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

define *P'* **where** $\langle P' \ i = \Phi (P \ i) \rangle$ **for** $i :: 'a$

have *proj-P'*: $\langle \text{is-Proj } (P' \ i) \rangle$ **for** i

by (*simp add: P-def P'-def register-projector*)

have $\langle \sum_{i \in \text{UNIV}} P \ i = \text{id-cblinfun} \rangle$

using *sum-butterfly-ket P-butter* **by** *simp*

then have *sumP'id*: $\langle \sum_{i \in \text{UNIV}} P' \ i = \text{id-cblinfun} \rangle$

unfolding *P'-def*

```

apply (subst complex-vector.linear-sum[OF ‹linear  $\Phi$ ›, symmetric])
by auto

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

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

obtain  $B$  where  $\text{ortho}B$ : ‹ $\text{is-ortho-set } (B\ i)$ ›
and  $\text{normal}B$ : ‹ $\bigwedge b. b \in B\ i \implies \text{norm } b = 1$ ›
and  $\text{cspan}B$ : ‹ $\text{cspan } (B\ i) = \text{cspan } (B0\ i)$ ›
and  $\text{finite}B$ : ‹ $\text{finite } (B\ i)$ › for  $i$ 
apply  $\text{atomize-elim}$  apply ( $\text{simp flip: all-conj-distrib}$ ) apply ( $\text{rule choice}$ )
using  $\text{orthonormal-basis-of-cspan}[OF\ \text{finite}B0]$  by blast

from  $\text{cspan}B\ \text{cspan}B0$  have  $\text{cspan}B$ : ‹ $\text{cspan } (B\ i) = \text{space-as-set } (S\ i)$ › for  $i$ 
by  $\text{simp}$ 
then have  $\text{ccspan}B$ : ‹ $\text{ccspan } (B\ i) = S\ i$ › for  $i$ 
by (metis  $\text{ccspan.rep-eq closure-finite-cspan finite}B\ \text{space-as-set-inject}$ )
from  $\text{ortho}B$  have  $\text{indep}B$ : ‹ $\text{cindependent } (B\ i)$ › for  $i$ 
by ( $\text{simp add: Complex-Inner-Product.is-ortho-set-cindependent}$ )

have  $\text{ortho}BiBj$ : ‹ $\text{is-orthogonal } x\ y$ › 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  $x$ : ‹ $x = P'\ i *_{\mathcal{V}} x'$ ›
by (metis  $S\text{-def cblinfun-fixes-range complex-vector.span-base cspan}B\ \text{is-Proj-idempotent proj-}P'$ )
from ‹ $y \in B\ j$ › obtain  $y'$  where  $y$ : ‹ $y = P'\ j *_{\mathcal{V}} y'$ ›
by (metis  $S\text{-def cblinfun-fixes-range complex-vector.span-base cspan}B\ \text{is-Proj-idempotent proj-}P'$ )
have ‹ $\text{cinner } x\ y = \text{cinner } (P'\ i *_{\mathcal{V}} x') (P'\ j *_{\mathcal{V}} y')$ ›
using  $x\ y$  by  $\text{simp}$ 
also have ‹ $\dots = \text{cinner } (P'\ j *_{\mathcal{V}} P'\ i *_{\mathcal{V}} x') y'$ ›
by (metis  $\text{cinner-adj-left is-Proj-algebraic proj-}P'$ )
also have ‹ $\dots = \text{cinner } (\Phi (P\ j\ o_{CL}\ P\ i) *_{\mathcal{V}} x') y'$ ›
unfolding  $P'\text{-def register-mult}[OF\ \langle\text{register } \Phi\rangle, \text{symmetric}]$  by  $\text{simp}$ 
also have ‹ $\dots = \text{cinner } (\Phi (\text{selfbutterket } j\ o_{CL}\ \text{selfbutterket } i) *_{\mathcal{V}} x') y'$ ›
unfolding  $P\text{-butter}$  by  $\text{simp}$ 
also have ‹ $\dots = \text{cinner } (\Phi\ 0 *_{\mathcal{V}} x') y'$ ›
by (metis  $\text{butterfly-comp-butterfly complex-vector.scale-eq-0-iff orthogonal-ket that}(3)$ )
also have ‹ $\dots = 0$ ›
by ( $\text{simp add: complex-vector.linear-0}$ )
finally show  $?thesis$ 
by –
qed

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

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

have ‹ $(\sum_{i \in UNIV} B'\ i) = \text{Proj } (\text{ccspan } B')$ ›
proof ( $\text{unfold } B'\text{-def, use finite}[of\ UNIV]$  in  $\text{induction}$ )
case  $\text{empty}$ 
show  $?case$  by auto
next
case ( $\text{insert } j\ M$ )
have ‹ $(\sum_{i \in \text{insert } j\ M} B'\ i) = P'\ j + (\sum_{i \in M} B'\ i)$ ›
by (meson  $\text{insert.hyps}(1)\ \text{insert.hyps}(2)\ \text{sum.insert}$ )

```



```

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 $\langle \dots = (\sum_{i \in UNIV}. \text{card } (B \ i)) \rangle$
unfolding B' -def **apply** (rule card-UN-disjoint)
using finiteB Bdisj **by** auto
also have $\langle \dots = (\sum_{(i::'a) \in UNIV}. \text{card } (B \ \xi 0)) \rangle$
using cardBsame **by** metis
also have $\langle \dots = \text{card } (B \ \xi 0) * \text{CARD}('a) \rangle$
by auto
finally show ?thesis **by** –
qed

obtain f **where** $\text{bij-}f$: $\langle \text{bij-betw } f \text{ (UNIV::('a,'b) complement-domain set) } (B \ \xi 0) \rangle$
apply atomize-elim **apply** (rule finite-same-card-bij)
using finiteB CARD-complement-domain[OF CARD'b] **by** auto

define u **where** $\langle u = (\lambda(\xi, \alpha). \Phi (\text{butterket } \xi \ \xi 0) *_{\vee} f \ \alpha) \rangle$ **for** $\xi :: 'a$ **and** $\alpha :: \langle ('a,'b) \text{ complement-domain} \rangle$
obtain U **where** $U\text{apply}$: $\langle U *_{\vee} \text{ket } \xi \alpha = u \ \xi \alpha \rangle$ **for** $\xi \alpha$
apply atomize-elim
apply (rule exI[of - $\langle \text{cblinfun-extension (range ket) } (\lambda k. u \text{ (inv ket } k) \rangle \rangle$])
apply (subst cblinfun-extension-apply)
apply (rule cblinfun-extension-exists-finite-dim)
by (auto simp add: inj-ket cindependent-ket)

define eqa **where** $\langle \text{eqa } a \ b = (\text{if } a = b \text{ then } 1 \text{ else } 0 :: \text{complex}) \rangle$ **for** $a \ b :: 'a$
define eqc **where** $\langle \text{eqc } a \ b = (\text{if } a = b \text{ then } 1 \text{ else } 0 :: \text{complex}) \rangle$ **for** $a \ b :: \langle ('a,'b) \text{ complement-domain} \rangle$
define eqac **where** $\langle \text{eqac } a \ b = (\text{if } a = b \text{ then } 1 \text{ else } 0 :: \text{complex}) \rangle$ **for** $a \ b :: \langle 'a * ('a,'b) \text{ complement-domain} \rangle$

have $\langle \text{cinner } (U *_{\vee} \text{ket } \xi \alpha) (U *_{\vee} \text{ket } \xi' \alpha') = \text{eqac } \xi \alpha \ \xi' \alpha' \rangle$ **for** $\xi \alpha \ \xi' \alpha'$
proof –
obtain $\xi \ \alpha \ \xi' \ \alpha'$ **where** $\xi \alpha$: $\langle \xi \alpha = (\xi, \alpha) \rangle$ **and** $\xi' \alpha'$: $\langle \xi' \alpha' = (\xi', \alpha') \rangle$
apply atomize-elim **by** auto
have $\langle \text{cinner } (U *_{\vee} \text{ket } (\xi, \alpha)) (U *_{\vee} \text{ket } (\xi', \alpha')) = \text{cinner } (\Phi (\text{butterket } \xi \ \xi 0) *_{\vee} f \ \alpha) (\Phi (\text{butterket } \xi' \ \xi 0) *_{\vee} f \ \alpha') \rangle$
unfolding $U\text{apply}$ u-def **by** simp
also have $\langle \dots = \text{cinner } ((\Phi (\text{butterket } \xi' \ \xi 0)) *_{\vee} \Phi (\text{butterket } \xi \ \xi 0) *_{\vee} f \ \alpha) (f \ \alpha') \rangle$
by (simp add: cinner-adj-left)
also have $\langle \dots = \text{cinner } (\Phi (\text{butterket } \xi' \ \xi 0) *) *_{\vee} \Phi (\text{butterket } \xi \ \xi 0) *_{\vee} f \ \alpha) (f \ \alpha') \rangle$
by (metis (no-types, lifting) assms register-def)
also have $\langle \dots = \text{cinner } (\Phi (\text{butterket } \xi 0 \ \xi' \ o_{CL} \ \text{butterket } \xi \ \xi 0) *_{\vee} f \ \alpha) (f \ \alpha') \rangle$
by (simp add: register-mult cblinfun-apply-cblinfun-compose[symmetric])
also have $\langle \dots = \text{cinner } (\Phi (\text{eqa } \xi' \ \xi *_{\vee} \text{selfbutterket } \xi 0) *_{\vee} f \ \alpha) (f \ \alpha') \rangle$
apply simp **by** (metis eqa-def cinner-ket)
also have $\langle \dots = \text{eqa } \xi' \ \xi * \text{cinner } (\Phi (\text{selfbutterket } \xi 0) *_{\vee} f \ \alpha) (f \ \alpha') \rangle$
by (smt (verit, ccfv-threshold) $\langle \text{linear } \Phi \rangle$ eqa-def cblinfun.scaleC-left cinner-commute cinner-scaleC-left cinner-zero-right complex-cnj-one complex-vector.linear-scale)
also have $\langle \dots = \text{eqa } \xi' \ \xi * \text{cinner } (P' \ \xi 0 *_{\vee} f \ \alpha) (f \ \alpha') \rangle$
using P-butter P'-def **by** simp
also have $\langle \dots = \text{eqa } \xi' \ \xi * \text{cinner } (f \ \alpha) (f \ \alpha') \rangle$
apply (subst P'id)
apply (metis bij-betw-imp-surj-on bij-f complex-vector.span-base cspanB rangeI)
by simp
also have $\langle \dots = \text{eqa } \xi' \ \xi * \text{eqc } \alpha \ \alpha' \rangle$
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 $\xi' \alpha' \ \xi \alpha$)
qed

then have $\langle \text{isometry } U \rangle$
apply (rule-tac orthogonal-on-basis-is-isometry[where $B = \langle \text{range ket} \rangle$])
using eqac-def **by** auto

have $\langle U *_{\vee} o_{CL} \Phi (\text{butterket } \xi \ \eta) o_{CL} U = \text{butterket } \xi \ \eta \otimes_{\circ} \text{id-cblinfun} \rangle$ **for** $\xi \ \eta$
proof (rule equal-ket, rename-tac $\xi 1 \alpha$)

fix $\xi 1\alpha$ **obtain** $\xi 1 :: 'a$ **and** $\alpha :: \langle ('a, 'b) \text{ complement-domain} \rangle$ **where** $\xi 1\alpha: \langle \xi 1\alpha = (\xi 1, \alpha) \rangle$
apply *atomize-elim* **by** *auto*
have $\langle (U *_{o_{CL}} \Phi (\text{butterket } \xi \eta)_{o_{CL}} U) *_{\mathcal{V}} \text{ket } \xi 1\alpha = U *_{\mathcal{V}} \Phi (\text{butterket } \xi \eta) *_{\mathcal{V}} \Phi (\text{butterket } \xi 1 \xi 0) *_{\mathcal{V}} f \alpha \rangle$
unfolding *cblinfun-apply-cblinfun-compose* $\xi 1\alpha$ *Uapply u-def* **by** *simp*
also have $\langle \dots = U *_{\mathcal{V}} \Phi (\text{butterket } \xi \eta)_{o_{CL}} \text{butterket } \xi 1 \xi 0 \rangle *_{\mathcal{V}} f \alpha$
by (*metis* *(no-types, lifting) assms butterfly-comp-butterfly lift-cblinfun-comp(4) register-mult*)
also have $\langle \dots = U *_{\mathcal{V}} \Phi (\text{eqa } \eta \xi 1 *_{\mathcal{C}} \text{butterket } \xi \xi 0) \rangle *_{\mathcal{V}} f \alpha$
by (*simp add: eqa-def cinner-ket*)
also have $\langle \dots = \text{eqa } \eta \xi 1 *_{\mathcal{C}} U *_{\mathcal{V}} \Phi (\text{butterket } \xi \xi 0) \rangle *_{\mathcal{V}} f \alpha$
by (*simp add: complex-vector.linear-scale*)
also have $\langle \dots = \text{eqa } \eta \xi 1 *_{\mathcal{C}} U *_{\mathcal{V}} U *_{\mathcal{V}} \text{ket } (\xi, \alpha) \rangle$
unfolding *Uapply u-def* **by** *simp*
also from *(isometry U)* **have** $\langle \dots = \text{eqa } \eta \xi 1 *_{\mathcal{C}} \text{ket } (\xi, \alpha) \rangle$
unfolding *cblinfun-apply-cblinfun-compose[symmetric]* **by** *simp*
also have $\langle \dots = (\text{butterket } \xi \eta) *_{\mathcal{V}} \text{ket } \xi 1 \rangle \otimes_s \text{ket } \alpha$
by (*simp add: eqa-def tensor-ell2-scaleC1*)
also have $\langle \dots = (\text{butterket } \xi \eta) \otimes_o \text{id-cblinfun} \rangle *_{\mathcal{V}} \text{ket } \xi 1\alpha$
by (*simp add: \xi 1\alpha tensor-op-ket*)
finally show $\langle (U *_{o_{CL}} \Phi (\text{butterket } \xi \eta)_{o_{CL}} U) *_{\mathcal{V}} \text{ket } \xi 1\alpha = (\text{butterket } \xi \eta) \otimes_o \text{id-cblinfun} \rangle *_{\mathcal{V}} \text{ket } \xi 1\alpha$
by –
qed
then have *1: \langle U *_{o_{CL}} \Phi \vartheta_{o_{CL}} U = \vartheta \otimes_o \text{id-cblinfun} \rangle* **for** ϑ
apply (*rule-tac clinear-eq-butterfly-ketI[THEN fun-cong, where x=\vartheta]*)
by (*auto intro!: clinearI simp add: bounded-cbilinear.add-left bounded-cbilinear-cblinfun-compose complex-vector.linear-add complex-vector.linear-scale*)

have *\langle unitary U \rangle*
proof –
have $\langle \Phi (\text{butterket } \xi \xi 1) *_{\mathcal{S}} \top \leq U *_{\mathcal{S}} \top \rangle$ **for** $\xi \xi 1$
proof –
have $*$: $\langle \Phi (\text{butterket } \xi \xi 0) *_{\mathcal{V}} b \in \text{space-as-set } (U *_{\mathcal{S}} \top) \rangle$ **if** $\langle b \in B \xi 0 \rangle$ **for** b
apply (*subst asm-rl[of \langle \Phi (\text{butterket } \xi \xi 0) *_{\mathcal{V}} b = u (\xi, \text{inv } f b) \rangle]*)
apply (*simp add: u-def, metis bij-betw-inv-into-right bij-f that*)
by (*metis Uapply cblinfun-apply-in-image*)

have $\langle \Phi (\text{butterket } \xi \xi 1) *_{\mathcal{S}} \top = \Phi (\text{butterket } \xi \xi 0) *_{\mathcal{S}} \Phi (\text{butterket } \xi 0 \xi 0) *_{\mathcal{S}} \Phi (\text{butterket } \xi 0 \xi 1) *_{\mathcal{S}} \top \rangle$
unfolding *cblinfun-compose-image[symmetric]* *register-mult[OF assms]*
by *simp*
also have $\langle \dots \leq \Phi (\text{butterket } \xi \xi 0) *_{\mathcal{S}} \Phi (\text{butterket } \xi 0 \xi 0) *_{\mathcal{S}} \top \rangle$
by (*meson cblinfun-image-mono top-greatest*)
also have $\langle \dots = \Phi (\text{butterket } \xi \xi 0) *_{\mathcal{S}} S \xi 0 \rangle$
by (*simp add: S-def P'-def P-butter*)
also have $\langle \dots = \Phi (\text{butterket } \xi \xi 0) *_{\mathcal{S}} \text{ccspan } (B \xi 0) \rangle$
by (*simp add: ccspanB*)
also have $\langle \dots = \text{ccspan } (\Phi (\text{butterket } \xi \xi 0) \text{ ` } B \xi 0) \rangle$
by (*meson cblinfun-image-ccspan*)
also have $\langle \dots \leq U *_{\mathcal{S}} \top \rangle$
by (*rule ccspan-leqI, use * in auto*)
finally show *?thesis* **by** –
qed
moreover have $\langle \Phi \text{id-cblinfun} *_{\mathcal{S}} \top \leq (\text{SUP } \xi \in \text{UNIV}. \Phi (\text{selfbutterket } \xi) *_{\mathcal{S}} \top) \rangle$
unfolding *sum-butterfly-ket[symmetric]*
apply (*subst complex-vector.linear-sum, simp*)
by (*rule cblinfun-sum-image-distr*)
ultimately have $\langle \Phi \text{id-cblinfun} *_{\mathcal{S}} \top \leq U *_{\mathcal{S}} \top \rangle$
apply *auto* **by** (*meson SUP-le-iff order.trans*)
then have $\langle U *_{\mathcal{S}} \top = \top \rangle$
apply *auto*
using *top.extremum-unique* **by** *blast*
with *(isometry U)* **show** *\langle unitary U \rangle*
by (*rule surj-isometry-is-unitary*)
qed

have $\langle \Phi \vartheta = U \circ_{CL} (\vartheta \otimes_o id\text{-cblinfun}) \circ_{CL} U^* \rangle$ **for** ϑ
proof –
from $\langle unitary\ U \rangle$
have $\langle \Phi \vartheta = (U \circ_{CL} U^*) \circ_{CL} \Phi \vartheta \circ_{CL} (U \circ_{CL} U^*) \rangle$
by *simp*
also have $\langle \dots = U \circ_{CL} (U^* \circ_{CL} \Phi \vartheta \circ_{CL} U) \circ_{CL} U^* \rangle$
by (*simp add: cblinfun-assoc-left*)
also have $\langle \dots = U \circ_{CL} (\vartheta \otimes_o id\text{-cblinfun}) \circ_{CL} U^* \rangle$
using 1 **by** *simp*
finally show *?thesis*
by –
qed

with $\langle unitary\ U \rangle$ **show** *?thesis*
by (*auto simp: sandwich-def*)
qed

lemma *register-decomposition-converse*:
assumes $\langle unitary\ U \rangle$
shows $\langle register\ (\lambda x. sandwich\ U\ (id\text{-cblinfun}\ \otimes_o\ x)) \rangle$
using - *unitary-sandwich-register* **apply** (*rule register-comp[unfolded o-def]*)
using *assms* **by** *auto*

lemma *register-inj*: $\langle inj\ F \rangle$ **if** $\langle register\ F \rangle$
proof –
obtain $U :: \langle ('a \times ('a, 'b)\ complement\ domain)\ ell2 \Rightarrow_{CL} 'b\ ell2 \rangle$
where $\langle unitary\ U \rangle$ **and** $F: \langle F\ a = sandwich\ U\ (a \otimes_o id\text{-cblinfun}) \rangle$ **for** a
apply *atomize-elim* **using** $\langle register\ F \rangle$ **by** (*rule register-decomposition*)
have $\langle inj\ (sandwich\ U) \rangle$
by (*smt (verit, best) <unitary U> cblinfun-assoc-left inj-onI sandwich-def cblinfun-compose-id-right cblinfun-compose-id-left unitary-def*)
moreover have $\langle inj\ (\lambda a::'a:finite\ ell2 \Rightarrow_{CL} \cdot.\ a \otimes_o id\text{-cblinfun}) \rangle$
by (*rule inj-tensor-left, simp*)
ultimately show $\langle inj\ F \rangle$
unfolding F
by (*smt (z3) inj-def*)
qed

lemma *iso-register-decomposition*:
assumes [*simp*]: $\langle iso\ register\ F \rangle$
shows $\langle \exists U. unitary\ U \wedge F = sandwich\ U \rangle$
proof –
have [*simp*]: $\langle register\ F \rangle$
using *assms iso-register-is-register* **by** *blast*

let $?ida = \langle id\text{-cblinfun} :: ('a, 'b)\ complement\ domain\ ell2 \Rightarrow_{CL} \cdot \rangle$

from *register-decomposition[OF <register F>]*
obtain $V :: \langle ('a \times ('a, 'b)\ complement\ domain)\ ell2 \Rightarrow_{CL} 'b\ ell2 \rangle$ **where** $\langle unitary\ V \rangle$
and $FV: \langle F\ \vartheta = sandwich\ V\ (\vartheta \otimes_o ?ida) \rangle$ **for** ϑ
by *auto*

have $\langle surj\ F \rangle$
by (*meson assms iso-register-inv-comp2 surj-iff*)
have *surj-tensor*: $\langle surj\ (\lambda a::'a\ ell2 \Rightarrow_{CL} 'a\ ell2.\ a \otimes_o ?ida) \rangle$
apply (*rule surj-from-comp[where g=<sandwich V>]*)
using $\langle surj\ F \rangle$ **apply** (*auto simp: FV*)
by (*meson <unitary V> register-inj unitary-sandwich-register*)
then obtain $a :: \langle 'a\ ell2 \Rightarrow_{CL} \cdot \rangle$
where $a: \langle a \otimes_o ?ida = selfbutterket\ undefined \otimes_o selfbutterket\ undefined \rangle$
by (*smt (verit, best) surjD*)

```

then have ‹a ≠ 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 ≠ 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 (λψ. ψ ⊗s ket undefined)› and ‹U = V oCL T›
have T: ‹T ψ = ψ ⊗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 ⊗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 oCL (butterfly x y ⊗o ?ida) oCL V*› for x y
  by (simp add: Misc.sandwich-def FV)
also have ‹... x y = V oCL (butterfly (T x) (T y)) oCL V*› for x y
  by (simp add: T γ ‹γ = 1›)
also have ‹... x y = U oCL (butterfly x y) oCL U*› for x y
  by (simp add: U-def butterfly-comp-cblinfun cblinfun-comp-butterfly)
finally have F-rep: ‹F a = U oCL a oCL 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 T = T›
proof -
  have 1: ‹φ ⊗s ξ ∈ range ((*V) T)› for φ ξ
  proof -
    have ‹T *V (cinner (ket undefined) ξ *C φ) = φ ⊗s (cinner (ket undefined) ξ *C ket undefined)›
      by (simp add: T tensor-ell2-scaleC2)
    also have ‹... = φ ⊗s (selfbutterket undefined *V ξ)›
      by simp
    also have ‹... = φ ⊗s (?ida *V ξ)›
      by (simp add: γ ‹γ = 1›)
    also have ‹... = φ ⊗s ξ›
      by simp
    finally show ?thesis
      by (metis range-eqI)
  qed
qed

have ‹T ≤ ccspan {ket x | x. True}›
  by (simp add: full-SetCompr-eq)
also have ‹... ≤ ccspan {φ ⊗s ξ | φ ξ. True}›
  apply (rule ccspan-mono)

```

```

    by (auto simp flip: tensor-ell2-ket)
  also from 1 have ⟨... ≤ cspan (range ((*V) T))⟩
    by (auto intro!: cspan-mono)
  also have ⟨... = T *S T⟩
  by (metis (mono-tags, opaque-lifting) calculation cblinfun-image-cspan cblinfun-image-mono eq-iff top-greatest)
  finally show ⟨T *S T = T⟩
    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 [[simploc 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)
  qed

```

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

definition $\langle commutant\ F = \{x. \forall y \in F. x\ o_{CL}\ y = y\ o_{CL}\ x\} \rangle$

lemma *commutant-exchange*:

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

lemma *commutant-tensor1*: $\langle commutant\ (range\ (\lambda a. a\ \otimes_o\ id-cblinfun)) = range\ (\lambda b. id-cblinfun\ \otimes_o\ b) \rangle$

proof $(rule\ Set.set-eqI,\ rule\ iffI)$

fix $x :: \langle ('a \times 'b)\ ell2 \Rightarrow_{CL}\ ('a \times 'b)\ ell2 \rangle$

fix $\gamma :: 'a$

assume $\langle x \in commutant\ (range\ (\lambda a. a\ \otimes_o\ id-cblinfun)) \rangle$

then have *comm*: $\langle (a\ \otimes_o\ id-cblinfun)\ *_V\ x\ *_V\ \psi = x\ *_V\ (a\ \otimes_o\ id-cblinfun)\ *_V\ \psi \rangle$ **for** $a\ \psi$

by $(metis\ (mono-tags,\ lifting)\ commutant-def\ mem-Collect-eq\ rangeI\ cblinfun-apply-cblinfun-compose)$

obtain x' **where** $x': \langle cinner\ (ket\ j)\ (x'\ *_V\ ket\ l) = cinner\ (ket\ (\gamma,j))\ (x\ *_V\ ket\ (\gamma,l)) \rangle$ **for** $j\ l$

proof *atomize-elim*

obtain ψ **where** $\psi: \langle cinner\ (ket\ j)\ (\psi\ l) = cinner\ (ket\ (\gamma,j))\ (x\ *_V\ ket\ (\gamma,l)) \rangle$ **for** $l\ j$

apply $(atomize-elim,\ rule\ choice,\ rule\ allI)$

apply $(rule-tac\ x=\langle Abs-ell2\ (\lambda j. cinner\ (ket\ (\gamma,j))\ (x\ *_V\ ket\ (\gamma,l))) \rangle$ **in** *exI*)

by $(simp\ add: cinner-ket-left\ Abs-ell2-inverse)$

obtain x' **where** $\langle x'\ *_V\ ket\ l = \psi\ l \rangle$ **for** l

apply *atomize-elim*

apply $(rule\ exI[of\ -\ \langle cblinfun-extension\ (range\ ket)\ (\lambda l. \psi\ (inv\ ket\ l)) \rangle])$

apply $(subst\ cblinfun-extension-apply)$

apply $(rule\ cblinfun-extension-exists-finite-dim)$

by $(auto\ simp\ add: inj-ket\ cindependent-ket)$

with ψ **have** $\langle cinner\ (ket\ j)\ (x'\ *_V\ ket\ l) = cinner\ (ket\ (\gamma,j))\ (x\ *_V\ ket\ (\gamma,l)) \rangle$ **for** $j\ l$

by *auto*

then show $\langle \exists x'. \forall j\ l. cinner\ (ket\ j)\ (x'\ *_V\ ket\ l) = cinner\ (ket\ (\gamma,j))\ (x\ *_V\ ket\ (\gamma,l)) \rangle$

by *auto*

qed

have $\langle cinner\ (ket\ (i,j))\ (x\ *_V\ ket\ (k,l)) = cinner\ (ket\ (i,j))\ ((id-cblinfun\ \otimes_o\ x')\ *_V\ ket\ (k,l)) \rangle$ **for** $i\ j\ k\ l$

proof $-$

have $\langle \text{cinner } (\text{ket } (i,j)) (x *_{\mathcal{V}} \text{ket } (k,l))$
 $= \text{cinner } ((\text{butterket } i \ \gamma \otimes_{\circ} \text{id-cblinfun}) *_{\mathcal{V}} \text{ket } (\gamma,j)) (x *_{\mathcal{V}} (\text{butterket } k \ \gamma \otimes_{\circ} \text{id-cblinfun}) *_{\mathcal{V}} \text{ket } (\gamma,l)) \rangle$
by (*auto simp: tensor-op-ket*)
also have $\langle \dots = \text{cinner } (\text{ket } (\gamma,j)) ((\text{butterket } \gamma \ i \otimes_{\circ} \text{id-cblinfun}) *_{\mathcal{V}} x *_{\mathcal{V}} (\text{butterket } k \ \gamma \otimes_{\circ} \text{id-cblinfun}) *_{\mathcal{V}} \text{ket } (\gamma,l)) \rangle$
by (*metis (no-types, lifting) cinner-adj-left butterfly-adjoint id-cblinfun-adjoint tensor-op-adjoint*)
also have $\langle \dots = \text{cinner } (\text{ket } (\gamma,j)) (x *_{\mathcal{V}} (\text{butterket } \gamma \ i \otimes_{\circ} \text{id-cblinfun } o_{CL} \text{ butterket } k \ \gamma \otimes_{\circ} \text{id-cblinfun}) *_{\mathcal{V}} \text{ket } (\gamma,l)) \rangle$
unfolding *comm* **by** (*simp add: cblinfun-apply-cblinfun-compose*)
also have $\langle \dots = \text{cinner } (\text{ket } i) (\text{ket } k) * \text{cinner } (\text{ket } (\gamma,j)) (x *_{\mathcal{V}} \text{ket } (\gamma,l)) \rangle$
by (*simp add: comp-tensor-op tensor-op-ket tensor-op-scaleC-left*)
also have $\langle \dots = \text{cinner } (\text{ket } i) (\text{ket } k) * \text{cinner } (\text{ket } j) (x' *_{\mathcal{V}} \text{ket } l) \rangle$
by (*simp add: x'*)
also have $\langle \dots = \text{cinner } (\text{ket } (i,j)) ((\text{id-cblinfun } \otimes_{\circ} x') *_{\mathcal{V}} \text{ket } (k,l)) \rangle$
apply (*simp add: tensor-op-ket*)
by (*simp flip: tensor-ell2-ket*)
finally show *?thesis* **by** –
qed
then have $\langle x = (\text{id-cblinfun } \otimes_{\circ} x') \rangle$
by (*auto intro!: equal-ket cinner-ket-eqI*)
then show $\langle x \in \text{range } (\lambda b. \text{id-cblinfun } \otimes_{\circ} b) \rangle$
by *auto*
next
fix $x :: \langle ('a \times 'b) \text{ ell2} \Rightarrow_{CL} ('a \times 'b) \text{ ell2} \rangle$
assume $\langle x \in \text{range } (\lambda b. \text{id-cblinfun } \otimes_{\circ} b) \rangle$
then obtain b **where** $x: \langle x = \text{id-cblinfun } \otimes_{\circ} b \rangle$
by *auto*
then show $\langle x \in \text{commutant } (\text{range } (\lambda a. a \otimes_{\circ} \text{id-cblinfun})) \rangle$
by (*auto simp: x commutant-def comp-tensor-op*)
qed

lemma *complement-range:*

assumes [*simp*]: $\langle \text{compatible } F \ G \rangle$ **and** [*simp*]: $\langle \text{iso-register } (F;G) \rangle$
shows $\langle \text{range } G = \text{commutant } (\text{range } F) \rangle$

proof –

have [*simp*]: $\langle \text{register } F \rangle \langle \text{register } G \rangle$
using *assms compatible-def* **by** *metis+*
have [*simp*]: $\langle (F;G) (a \otimes_{\circ} b) = F \ a \ o_{CL} \ G \ b \rangle$ **for** $a \ b$
using *Laws-Quantum.register-pair-apply assms* **by** *blast*
have [*simp*]: $\langle \text{range } F = (F;G) \text{ ' range } (\lambda a. a \otimes_{\circ} \text{id-cblinfun}) \rangle$
by *force*
have [*simp*]: $\langle \text{range } G = (F;G) \text{ ' range } (\lambda b. \text{id-cblinfun } \otimes_{\circ} b) \rangle$
by *force*
show $\langle \text{range } G = \text{commutant } (\text{range } F) \rangle$
by (*simp add: commutant-exchange commutant-tensor1*)

qed

lemma *same-range-equivalent:*

fixes $F :: \langle 'a::\text{finite update} \Rightarrow 'c::\text{finite update} \rangle$ **and** $G :: \langle 'b::\text{finite update} \Rightarrow 'c::\text{finite update} \rangle$
assumes [*simp*]: $\langle \text{register } F \rangle$ **and** [*simp*]: $\langle \text{register } G \rangle$
assumes $\langle \text{range } F = \text{range } G \rangle$
shows $\langle \text{equivalent-registers } F \ G \rangle$

proof –

have $G\text{-range}F$ [*simp*]: $\langle G \ x \in \text{range } F \rangle$ **for** x
by (*simp add: assms*)
have $F\text{-range}G$ [*simp*]: $\langle F \ x \in \text{range } G \rangle$ **for** x
by (*simp add: assms(3)[symmetric]*)
have [*simp*]: $\langle \text{inj } F \rangle$ **and** [*simp*]: $\langle \text{inj } G \rangle$
by (*simp-all add: register-inj*)
have [*simp*]: $\langle \text{clinear } F \rangle \langle \text{clinear } G \rangle$
by *simp-all*
define $I \ J$ **where** $\langle I \ x = \text{inv } F (G \ x) \rangle$ **and** $\langle J \ y = \text{inv } G (F \ y) \rangle$ **for** $x \ y$
have *addI*: $\langle I (x + y) = I \ x + I \ y \rangle$ **for** $x \ y$


```

unfolding I-def
apply (rule injD[OF ⟨inj F⟩])
apply (subst complex-vector.linear-add[OF ⟨linear 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 ⟨linear 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 ⟨linear 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 ⟨linear 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 oCL b) = I a oCL 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 oCL b) = J a oCL 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  $\langle \text{register } J \rangle$ 
  unfolding register-def by (auto intro!; clinearI)

have  $\langle I \circ J = \text{id} \rangle$ 
  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  $\langle J \circ I = \text{id} \rangle$ 
  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  $\langle I \circ J = \text{id} \rangle \langle J \circ I = \text{id} \rangle \langle \text{register } I \rangle \langle \text{register } J \rangle$ 
have  $\langle \text{iso-register } I \rangle$ 
  using iso-register-def by blast

have  $\langle F \circ I = G \rangle$ 
  unfolding I-def o-def
  by (subst Hilbert-Choice.f-inv-into-f[where f=F], auto)

with  $\langle \text{iso-register } I \rangle$  show ?thesis
  unfolding equivalent-registers-def by auto
qed

lemma complement-unique:
  assumes compatible F G and  $\langle \text{iso-register } (F;G) \rangle$ 
  assumes compatible F H and  $\langle \text{iso-register } (F;H) \rangle$ 
  shows  $\langle \text{equivalent-registers } G H \rangle$ 
  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  $\langle \text{complements } F G \iff \text{compatible } F G \wedge \text{iso-register } (F;G) \rangle$ 

lemma complementsI:  $\langle \text{compatible } F G \implies \text{iso-register } (F;G) \implies \text{complements } F G \rangle$ 
  using complements-def by blast

lemma complements-sym:  $\langle \text{complements } G F \rangle$  if  $\langle \text{complements } F G \rangle$ 
proof (rule complementsI)
  show [simp]:  $\langle \text{compatible } G F \rangle$ 
    using compatible-sym complements-def that by blast
  from that have  $\langle \text{iso-register } (F;G) \rangle$ 
    by (meson complements-def)
  then obtain I where [simp]:  $\langle \text{register } I \rangle$  and  $\langle (F;G) \circ I = \text{id} \rangle$  and  $\langle I \circ (F;G) = \text{id} \rangle$ 
    using iso-register-def by blast
  have  $\langle \text{register } (\text{swap } \circ I) \rangle$ 
    using  $\langle \text{register } I \rangle$  register-comp register-swap by blast
  moreover have  $\langle (G;F) \circ (\text{swap } \circ I) = \text{id} \rangle$ 
    by (simp add: <(F;G) o I = id> rewriteL-comp-comp)
  moreover have  $\langle (\text{swap } \circ I) \circ (G;F) = \text{id} \rangle$ 
    by (metis (no-types, opaque-lifting) swap-swap <I o (F;G) = id> calculation(2) comp-def eq-id-iff)

```

ultimately show $\langle \text{iso-register } (G;F) \rangle$
using $\langle \text{compatible } G \ F \rangle$ iso-register-def pair-is-register **by** blast
qed

definition $\text{complement} :: \langle ('a::\text{finite update} \Rightarrow 'b::\text{finite update}) \Rightarrow (('a,'b) \text{ complement-domain update} \Rightarrow 'b \text{ update}) \rangle$ **where**
 $\langle \text{complement } F = (\text{SOME } G :: ('a, 'b) \text{ complement-domain update} \Rightarrow 'b \text{ update. compatible } F \ G \wedge \text{iso-register } (F;G)) \rangle$

lemma $\text{register-complement}[simp]: \langle \text{register } (\text{complement } F) \rangle$ **if** $\langle \text{register } F \rangle$
using $\text{complement-exists}[OF \ \text{that}]$
by $(\text{metis } (\text{no-types, lifting}) \text{ compatible-def complement-def some-eq-imp})$

lemma $\text{complement-is-complement}:$
assumes $\langle \text{register } F \rangle$
shows $\langle \text{complements } F \ (\text{complement } F) \rangle$
using $\text{complement-exists}[OF \ \text{assms}]$ **unfolding** complements-def
by $(\text{metis } (\text{mono-tags, lifting}) \text{ complement-def some-eq-imp})$

lemma $\text{complement-unique}:$
assumes $\langle \text{complements } F \ G \rangle$
shows $\langle \text{equivalent-registers } G \ (\text{complement } F) \rangle$
apply $(\text{rule } \text{complement-unique}[\text{where } F=F])$
using assms **unfolding** complements-def **using** $\text{compatible-register1}$ $\text{complement-is-complement}$ complements-def
by blast+

lemma $\text{compatible-complement}[simp]: \langle \text{register } F \Longrightarrow \text{compatible } F \ (\text{complement } F) \rangle$
using $\text{complement-is-complement}$ complements-def **by** blast

lemma $\text{complements-register-tensor}:$
assumes $[simp]: \langle \text{register } F \rangle \langle \text{register } G \rangle$
shows $\langle \text{complements } (F \otimes_r G) \ (\text{complement } F \otimes_r \text{complement } G) \rangle$

proof $(\text{rule } \text{complementsI})$

have $\text{sep4}: \langle \text{separating } \text{TYPE}('z::\text{finite}) \ \{(a \otimes_u b) \otimes_u (c \otimes_u d) \mid a \ b \ c \ d. \ \text{True}\} \rangle$
apply $(\text{rule } \text{separating-tensor}'[\text{where } A=\langle \{(a \otimes_u b) \mid a \ b. \ \text{True}\} \rangle \ \text{and } B=\langle \{(c \otimes_u d) \mid c \ d. \ \text{True}\} \rangle])$
apply $(\text{rule } \text{separating-tensor}'[\text{where } A=\text{UNIV} \ \text{and } B=\text{UNIV}])$ **apply** $\text{auto}[3]$
apply $(\text{rule } \text{separating-tensor}'[\text{where } A=\text{UNIV} \ \text{and } B=\text{UNIV}])$ **apply** $\text{auto}[3]$
by auto

show $\text{compat}: \langle \text{compatible } (F \otimes_r G) \ (\text{complement } F \otimes_r \text{complement } G) \rangle$
by $(\text{metis } \text{assms}(1) \ \text{assms}(2) \ \text{compatible-register-tensor} \ \text{complement-is-complement} \ \text{complements-def})$

let $\text{?reorder} = \langle ((Fst \ o \ Fst; \ Snd \ o \ Fst); (Fst \ o \ Snd; \ Snd \ o \ Snd)) \rangle$

have $[simp]: \langle \text{register } \text{?reorder} \rangle$

by auto

have $[simp]: \langle \text{?reorder } ((a \otimes_u b) \otimes_u (c \otimes_u d)) = ((a \otimes_u c) \otimes_u (b \otimes_u d)) \rangle$
for $a::'t::\text{finite update}$ **and** $b::'u::\text{finite update}$ **and** $c::'v::\text{finite update}$ **and** $d::'w::\text{finite update}$
by $(\text{simp } \text{add: register-pair-apply } Fst\text{-def } Snd\text{-def } \text{comp-tensor-op})$

have $[simp]: \langle \text{iso-register } \text{?reorder} \rangle$

apply $(\text{rule } \text{iso-registerI}[of \ - \ \text{?reorder}])$ **apply** $\text{auto}[2]$

apply $(\text{rule } \text{register-eqI}[OF \ \text{sep4}])$ **apply** $\text{auto}[3]$

apply $(\text{rule } \text{register-eqI}[OF \ \text{sep4}])$ **by** auto

have $\langle (F \otimes_r G; \text{complement } F \otimes_r \text{complement } G) = ((F; \text{complement } F) \otimes_r (G; \text{complement } G)) \ o \ \text{?reorder} \rangle$

apply $(\text{rule } \text{register-eqI}[OF \ \text{sep4}])$

by $(\text{auto } \text{intro!}: \text{register-preregister } \text{register-comp } \text{register-tensor-is-register } \text{pair-is-register})$

$\text{simp: compat } \text{register-pair-apply } \text{comp-tensor-op}$

moreover **have** $\langle \text{iso-register } \dots \rangle$

apply $(\text{auto } \text{intro!}: \text{iso-register-comp } \text{iso-register-tensor-is-iso-register})$

using assms $\text{complement-is-complement}$ complements-def **by** blast+

ultimately show $\langle \text{iso-register } (F \otimes_r G; \text{complement } F \otimes_r \text{complement } G) \rangle$

by simp

qed

definition is-unit-register **where**

$\langle \text{is-unit-register } U \longleftrightarrow \text{complements } U \ \text{id} \rangle$

lemma *register-unit-register*[simp]: $\langle \text{is-unit-register } U \implies \text{register } U \rangle$
by (*simp add: compatible-def complements-def is-unit-register-def*)

lemma *unit-register-compatible*[simp]: $\langle \text{compatible } U \ X \rangle$ **if** $\langle \text{is-unit-register } U \rangle$ $\langle \text{register } X \rangle$
by (*metis compatible-comp-right complements-def id-comp is-unit-register-def that(1) that(2)*)

lemma *unit-register-compatible'*[simp]: $\langle \text{compatible } X \ U \rangle$ **if** $\langle \text{is-unit-register } U \rangle$ $\langle \text{register } X \rangle$
using *compatible-sym that(1) that(2) unit-register-compatible* **by** *blast*

lemma *compatible-complement-left*[simp]: $\langle \text{register } X \implies \text{compatible } (\text{complement } X) \ X \rangle$
using *compatible-sym complement-is-complement complements-def* **by** *blast*

lemma *compatible-complement-right*[simp]: $\langle \text{register } X \implies \text{compatible } X \ (\text{complement } X) \rangle$
using *complement-is-complement complements-def* **by** *blast*

lemma *unit-register-pair*[simp]: $\langle \text{equivalent-registers } X \ (U; \ X) \rangle$ **if** [simp]: $\langle \text{is-unit-register } U \rangle$ $\langle \text{register } X \rangle$
proof –
have $\langle \text{equivalent-registers } \text{id} \ (U; \ \text{id}) \rangle$
using *complements-def is-unit-register-def iso-register-equivalent-id that(1)* **by** *blast*
also have $\langle \text{equivalent-registers } \dots \ (U; \ (X; \ \text{complement } X)) \rangle$
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 $\langle \text{equivalent-registers } \dots \ ((U; \ X); \ \text{complement } X) \rangle$
apply (*rule equivalent-registers-assoc*)
by *auto*
finally have $\langle \text{complements } (U; \ X) \ (\text{complement } X) \rangle$
by (*auto simp: equivalent-registers-def complements-def*)
moreover have $\langle \text{equivalent-registers } (X; \ \text{complement } X) \ \text{id} \rangle$
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]: $\langle \text{is-unit-register } U \rangle$
assumes [simp]: $\langle \text{register } A \rangle$
shows $\langle \text{is-unit-register } (A \ o \ U) \rangle$
proof –
have $\langle \text{compatible } (A \ o \ U) \ (A; \ \text{complement } A) \rangle$
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]: $\langle \text{compatible } (A \ o \ U) \ \text{id} \rangle$
by (*metis assms(2) compatible-comp-right complement-is-complement complements-def iso-register-def*)
have $\langle \text{equivalent-registers } (A \ o \ U; \ \text{id}) \ (A \ o \ U; \ (A; \ \text{complement } A)) \rangle$
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 $\langle \text{equivalent-registers } \dots \ ((A \ o \ U; \ A \ o \ \text{id}); \ \text{complement } A) \rangle$
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 $\langle \text{equivalent-registers } \dots \ (A \ o \ (U; \ \text{id}); \ \text{complement } A) \rangle$
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 $\langle \text{equivalent-registers } \dots \ (A \ o \ \text{id}; \ \text{complement } A) \rangle$
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 $\langle \text{equivalent-registers } \dots \ \text{id} \rangle$
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 o A) id⟩
    by (simp add: iso-register-is-register)
  have 1: ⟨iso-register ((U;id) o 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-register)
  have 2: ⟨id o ((U;id) o A ⊗r id) = (U o 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 o 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 ((λd. tensor-op id-cblinfun d) o G)⟩
    by (simp add: assms(2) unit-register-compose-left)
  moreover have ⟨is-unit-register ((λc. tensor-op c id-cblinfun) o F)⟩
    using assms(1) register-tensor-left unit-register-compose-left by blast
  ultimately have ⟨equivalent-registers ((λd. tensor-op id-cblinfun d) o G) ((λc. 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 compatible.3' 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 o G;(complement F;F o complement G)) (F o G;(F o 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 o G;F o 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 o G;(complement F;F o 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  $\implies$  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  $\implies$  is-unit-register U  $\implies$  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  $\Rightarrow$  '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  $x = \langle id-cblinfun \otimes_o - \rangle$ ]
    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 $\langle \text{pure-state-target-vector } F \ \eta_F = (\text{if } \text{ket default} \in \text{range } (\text{cblinfun-apply } (F \ (\text{butterfly } \ \eta_F \ \eta_F))))$
then ket default
else $(\text{SOME } \eta'. \ \text{norm } \eta' = 1 \wedge \eta' \in \text{range } (\text{cblinfun-apply } (F \ (\text{butterfly } \ \eta_F \ \eta_F)))) \rangle$

lemma *pure-state-target-vector-eqI:*

assumes $\langle F \ (\text{butterfly } \ \eta_F \ \eta_F) = G \ (\text{butterfly } \ \eta_G \ \eta_G) \rangle$
shows $\langle \text{pure-state-target-vector } F \ \eta_F = \text{pure-state-target-vector } G \ \eta_G \rangle$
by (*simp add: assms pure-state-target-vector-def*)

lemma *pure-state-target-vector-ket-default:* $\langle \text{pure-state-target-vector } F \ \eta_F = \text{ket default} \ \mathbf{if} \ \langle \text{ket default} \in \text{range} \ (\text{cblinfun-apply } (F \ (\text{butterfly } \ \eta_F \ \eta_F))) \rangle \rangle$

by (*simp add: pure-state-target-vector-def that*)

lemma

assumes [*simp*]: $\langle \eta_F \neq 0 \rangle \langle \text{register } F \rangle$

shows *pure-state-target-vector-in-range:* $\langle \text{pure-state-target-vector } F \ \eta_F \in \text{range } ((*_V) \ (F \ (\text{selfbutter } \ \eta_F))) \rangle$ (**is** *?range*)

and *pure-state-target-vector-norm:* $\langle \text{norm } (\text{pure-state-target-vector } F \ \eta_F) = 1 \rangle$ (**is** *?norm*)

proof –

from *assms* **have** $\langle \text{selfbutter } \ \eta_F \neq 0 \rangle$

by (*metis butterfly-0-right complex-vector.scale-zero-right inj-selfbutter-upto-phase*)

then **have** $\langle F \ (\text{selfbutter } \ \eta_F) \neq 0 \rangle$

using *register-inj[OF register F], THEN injD, where y=0]*

by (*auto simp: complex-vector.linear-0*)

then **obtain** ψ' **where** $\psi': \langle F \ (\text{selfbutter } \ \eta_F) *_V \ \psi' \neq 0 \rangle$

by (*meson cblinfun-eq-0-on-UNIV-span complex-vector.span-UNIV*)

have *ex:* $\langle \exists \psi. \ \text{norm } \psi = 1 \wedge \psi \in \text{range } ((*_V) \ (F \ (\text{selfbutter } \ \eta_F))) \rangle$

apply (*rule exI[of - $\langle (F \ (\text{selfbutter } \ \eta_F) *_V \ \psi') /_C \ \text{norm } (F \ (\text{selfbutter } \ \eta_F) *_V \ \psi') \rangle$]*)

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*]: $\langle \eta \neq 0 \rangle \langle \text{register } F \rangle$

shows $\langle F \ (\text{selfbutter } \ \eta) *_V \ \text{pure-state-target-vector } F \ \eta \neq 0 \rangle$

proof –

obtain ψ **where** $\psi: \langle F \ (\text{selfbutter } \ \eta) \ \psi = \text{pure-state-target-vector } F \ \eta \rangle$

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** $\langle n = \text{cinner } \ \eta \ \eta \rangle$

then **have** $\langle n \neq 0 \rangle$

by *auto*

have *pure-state-target-vector-neq0:* $\langle \text{pure-state-target-vector } F \ \eta \neq 0 \rangle$

using *pure-state-target-vector-norm[OF assms]*

by *auto*

have $\langle F \ (\text{selfbutter } \ \eta) *_V \ \text{pure-state-target-vector } F \ \eta = F \ (\text{selfbutter } \ \eta) *_V \ F \ (\text{selfbutter } \ \eta) *_V \ \psi \rangle$

by (*simp add: ψ*)

also **have** $\langle \dots = n *_C \ F \ (\text{selfbutter } \ \eta) *_V \ \psi \rangle$

by (*simp flip: cblinfun-apply-cblinfun-compose add: register-mult register-scaleC n-def*)

also **have** $\langle \dots = n *_C \ \text{pure-state-target-vector } F \ \eta \rangle$

by (*simp add: ψ*)
 also have $\langle \dots \neq 0 \rangle$
 using *pure-state-target-vector-neq0* $\langle n \neq 0 \rangle$
 by *auto*
 finally show *?thesis*
 by –
 qed

definition $\langle \text{pure-state}' F \psi \eta_F = F (\text{butterfly } \psi \eta_F) *_V \text{pure-state-target-vector } F \eta_F \rangle$

abbreviation $\langle \text{pure-state } F \psi \equiv \text{pure-state}' F \psi (\text{ket default}) \rangle$

nonterminal *pure-tensor*

syntax *-pure-tensor* :: $\langle 'a \Rightarrow 'b \Rightarrow \text{pure-tensor} \Rightarrow \text{pure-tensor} \rangle (- - \otimes_p - [1000, 0, 0] 1000)$

syntax *-pure-tensor2* :: $\langle 'a \Rightarrow 'b \Rightarrow 'c \Rightarrow 'd \Rightarrow \text{pure-tensor} \rangle (- - \otimes_p - - [1000, 0, 1000, 0] 1000)$

syntax *-pure-tensor1* :: $\langle 'a \Rightarrow 'b \Rightarrow \text{pure-tensor} \rangle$

syntax *-pure-tensor-start* :: $\langle \text{pure-tensor} \Rightarrow 'a \rangle (('(-'))$

translations

-pure-tensor2 $F \psi G \varphi \rightarrow \text{CONST pure-state } (F; G) (\psi \otimes_s \varphi)$

-pure-tensor $F \psi (\text{CONST pure-state } G \varphi) \rightarrow \text{CONST pure-state } (F; G) (\psi \otimes_s \varphi)$

-pure-tensor-start $x \rightarrow x$

-pure-tensor-start $(\text{-pure-tensor2 } F \psi G \varphi) \leftarrow \text{CONST pure-state } (F; G) (\psi \otimes_s \varphi)$

-pure-tensor $F \psi (\text{-pure-tensor2 } G \varphi H \eta) \leftarrow \text{-pure-tensor2 } F \psi (G; H) (\varphi \otimes_s \eta)$

term $\langle (F \psi \otimes_p G \varphi \otimes_p H z) \rangle$

term $\langle \text{pure-state } (F; G) (a \otimes_s b) \rangle$

lemma *register-pair-butterfly-tensor*: $\langle (F; G) (\text{butterfly } (a \otimes_s b) (c \otimes_s d)) = F (\text{butterfly } a c) \circ_{CL} G (\text{butterfly } b d) \rangle$

if [*simp*]: $\langle \text{compatible } F G \rangle$

by (*auto simp: default-prod-def simp flip: tensor-ell2-ket tensor-butterfly register-pair-apply*)

lemma *pure-state-eqI*:

assumes $\langle F (\text{selfbutter } \eta_F) = G (\text{selfbutter } \eta_G) \rangle$

assumes $\langle F (\text{butterfly } \psi \eta_F) = G (\text{butterfly } \varphi \eta_G) \rangle$

shows $\langle \text{pure-state}' F \psi \eta_F = \text{pure-state}' G \varphi \eta_G \rangle$

proof –

from *assms(1)* **have** $\langle \text{pure-state-target-vector } F \eta_F = \text{pure-state-target-vector } G \eta_G \rangle$

by (*rule pure-state-target-vector-eqI*)

with *assms(2)*

show *?thesis*

unfolding *pure-state'-def*

by *simp*

qed

definition $\langle \text{regular-register } F \longleftrightarrow \text{register } F \wedge (\exists a. (F; \text{complement } F) (\text{selfbutterket default } \otimes_o a) = \text{selfbutterket default}) \rangle$

lemma *regular-registerI*:

assumes [*simp*]: $\langle \text{register } F \rangle$

assumes [*simp*]: $\langle \text{complements } F G \rangle$

assumes *eq*: $\langle (F; G) (\text{selfbutterket default } \otimes_o a) = \text{selfbutterket default} \rangle$

shows $\langle \text{regular-register } F \rangle$

proof –

have [*simp*]: $\langle \text{compatible } F G \rangle$

using *assms* by (*simp add: complements-def*)

from $\langle \text{complements } F G \rangle$

obtain *I* **where** *cFI*: $\langle \text{complement } F \circ I = G \rangle$ **and** $\langle \text{iso-register } I \rangle$

apply *atomize-elim*

by (*meson Laws-Complement-Quantum.complement-unique equivalent-registers-def equivalent-registers-sym*)

```

have ⟨(F; complement F) (selfbuttket default ⊗o I a) = (F; G) (selfbuttket default ⊗o a)⟩
  using cFI by (auto simp: register-pair-apply)
also have ⟨... = selfbuttket 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) (selfbuttket default) = selfbuttket 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) (selfbuttket default) = selfbuttket default ⊗o aG⟩
    by (metis assms(3) complement-is-complement complements-def inj-iff inv-f-f iso-register-inv-comp1 regular-register-def)
  define t1 where ⟨t1 = inv ((F;G); complement (F;G)) (selfbuttket default)⟩
  define t2 where ⟨t2 = inv (F; (G; complement (F;G))) (selfbuttket default)⟩
  define t3 where ⟨t3 = inv (G; (F; complement (F;G))) (selfbuttket 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)) (selfbuttket default)⟩
    unfolding t2-def I
    apply (subst o-inv-distrib[symmetric])
    by (auto simp: pair-o-tensor)
  also have ⟨... = (selfbuttket 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 = selfbuttket 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 ⊗r J)⟩
  by (rule iso-register-bij, simp)
have [simp]: ⟨inv (id ⊗r J) = id ⊗r inv J⟩
  by auto

have ⟨t3 = (inv (id ⊗r J) o inv (G;complement G)) (selfbutterket default)⟩
  unfolding t3-def I
  apply (subst o-inv-distrib[symmetric])
  by (auto simp: pair-o-tensor)
also have ⟨... = (selfbutterket default ⊗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 ⊗o inv J aG⟩
  by simp

have *: ⟨((F;G); complement (F;G)) o assoc' = (F; (G; complement (F;G)))⟩
  apply (rule tensor-extensionality3)
  by (auto simp: register-pair-apply compatible-complement-pair1 compatible-complement-pair2)
have t2-t1: ⟨t2 = assoc t1⟩
  unfolding t1-def t2-def *[symmetric] apply (subst o-inv-distrib)
  by auto

have *: ⟨((F;G); complement (F;G)) o (swap ⊗r id) o assoc' = (G; (F; complement (F;G)))⟩
  apply (rule tensor-extensionality3)
  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 ⊗r id) t1)⟩
  unfolding t1-def t3-def *[symmetric] apply (subst o-inv-distrib)
  by (auto intro!: bij-comp simp: iso-register-bij o-inv-distrib)

from ⟨t2 = assoc t1⟩ ⟨t3 = assoc ((swap ⊗r id) t1)⟩
have *: ⟨selfbutterket default ⊗o inv J aG = assoc ((swap ⊗r id) (assoc' (selfbutterket default ⊗o inv I aF)))⟩
  by (simp add: t2' t3')

have ⟨selfbutterket default ⊗o swap (inv J aG) = (id ⊗r swap) (selfbutterket default ⊗o inv J aG)⟩
  by auto
also have ⟨... = ((id ⊗r swap) o assoc o (swap ⊗r id) o assoc') (selfbutterket default ⊗o inv I aF)⟩
  by (simp add: *)
also have ⟨... = (assoc o swap) (selfbutterket default ⊗o inv I aF)⟩
  apply (rule fun-cong[where g=⟨assoc o swap⟩])
  apply (intro tensor-extensionality3 register-comp register-tensor-is-register)
  by auto
also have ⟨... = assoc (inv I aF ⊗o selfbutterket default)⟩
  by auto
finally have *: ⟨selfbutterket default ⊗o swap (inv J aG) = assoc (inv I aF ⊗o selfbutterket default)⟩
  by -

obtain c where *: ⟨selfbutterket (default::'c) ⊗o swap (inv J aG) = selfbutterket default ⊗o c ⊗o selfbutterket
default⟩
  apply atomize-elim
  apply (rule overlapping-tensor)
  using * unfolding assoc-ell2-sandwich sandwich-def
  by auto

have ⟨t1 = ((swap ⊗r id) o assoc') t3⟩
  by (simp add: t3-t1 register-tensor-distrib[unfolded o-def, THEN fun-cong] flip: id-def)
also have ⟨... = ((swap ⊗r id) o assoc' o (id ⊗r swap)) (selfbutterket (default::'c) ⊗o swap (inv J aG))⟩
  unfolding t3' by auto
also have ⟨... = ((swap ⊗r id) o assoc' o (id ⊗r swap)) (selfbutterket default ⊗o c ⊗o selfbutterket default)⟩
  unfolding * by simp

```

also have $\langle \dots = \text{selfbutterket default } \otimes_o c \rangle$
apply (*simp del: tensor-butterfly*)
by (*simp add: default-prod-def*)
finally have $\langle t1 = \text{selfbutterket default } \otimes_o c \rangle$
by –

then show *?thesis*
apply (*auto intro!: exI[of - c] simp: regular-register-def t1-def*)
by (*metis* $\langle \text{bij } ((F;G); \text{complement } (F;G)) \rangle \text{bij-inv-eq-iff}$)
qed

lemma *regular-register-comp*: $\langle \text{regular-register } (F \circ G) \rangle$ **if** $\langle \text{regular-register } F \rangle$ $\langle \text{regular-register } G \rangle$
proof –
have [*simp*]: $\langle \text{register } F \rangle$ $\langle \text{register } G \rangle$
using *regular-register-def* **that** **by** *blast+*
from that obtain *a* **where** *a*: $\langle (F; \text{complement } F) (\text{selfbutterket default } \otimes_o a) = \text{selfbutterket default} \rangle$
unfolding *regular-register-def* **by** *metis*
from that obtain *b* **where** *b*: $\langle (G; \text{complement } G) (\text{selfbutterket default } \otimes_o b) = \text{selfbutterket default} \rangle$
unfolding *regular-register-def* **by** *metis*
have $\langle \text{complements } (F \circ G) (\text{complement } F; F \circ \text{complement } G) \rangle$
by (*simp add: complements-chain*)
then have $\langle \text{equivalent-registers } (\text{complement } F; F \circ \text{complement } G) (\text{complement } (F \circ G)) \rangle$
using *complement-unique* **by** *blast*
then obtain *J* **where** [*simp*]: $\langle \text{iso-register } J \rangle$ **and** *1*: $\langle (\text{complement } F; F \circ \text{complement } G) \circ J = (\text{complement } (F \circ G)) \rangle$
using *equivalent-registers-def* **by** *blast*
have [*simp*]: $\langle \text{register } J \rangle$
by (*simp add: iso-register-is-register*)

define *c* **where** $\langle c = \text{inv } J (a \otimes_o b) \rangle$

have $\langle ((F \circ G); \text{complement } (F \circ G)) (\text{selfbutterket default } \otimes_o c) = ((F \circ G); (\text{complement } F; F \circ \text{complement } G)) (\text{selfbutterket default } \otimes_o J c) \rangle$
by (*auto simp flip: 1 simp: register-pair-apply*)
also have $\langle \dots = ((F \circ (G; \text{complement } G); \text{complement } F) \circ \text{assoc}' \circ (\text{id } \otimes_r \text{swap})) (\text{selfbutterket default } \otimes_o J c) \rangle$
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 $\langle \dots = ((F \circ (G; \text{complement } G); \text{complement } F) \circ \text{assoc}') (\text{selfbutterket default } \otimes_o \text{swap } (J c)) \rangle$
by *auto*
also have $\langle \dots = ((F \circ (G; \text{complement } G); \text{complement } F) \circ \text{assoc}') (\text{selfbutterket default } \otimes_o (b \otimes_o a)) \rangle$
unfolding *c-def* **apply** (*subst surj-f-inv-f[where f=J]*)
apply (*meson* $\langle \text{iso-register } J \rangle \text{bij-betw-inv-into-right iso-register-inv-comp1 iso-register-inv-comp2 iso-tuple-UNIV-I o-bij surj-iff-all}$)
by *auto*
also have $\langle \dots = (F \circ (G; \text{complement } G); \text{complement } F) ((\text{selfbutterket default } \otimes_o b) \otimes_o a) \rangle$
by (*simp add: assoc'-apply*)
also have $\langle \dots = (F; \text{complement } F) ((G; \text{complement } G) (\text{selfbutterket default } \otimes_o b) \otimes_o a) \rangle$
by (*simp add: register-pair-apply'*)
also have $\langle \dots = \text{selfbutterket default} \rangle$
by (*auto simp: a b*)
finally have $\langle (F \circ G; \text{complement } (F \circ G)) (\text{selfbutterket default } \otimes_o c) = \text{selfbutterket default} \rangle$
by –
then show *?thesis*
using $\langle \text{register } F \rangle$ $\langle \text{register } G \rangle$ *register-comp regular-register-def* **by** *blast*
qed

lemma *regular-iso-register*:
assumes $\langle \text{regular-register } F \rangle$

```

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 o I⟩ a by presburger
also have ⟨... = (F; ?u) (selfbutterket default ⊗o I a)⟩
by (metis Laws-Quantum.register-pair-apply ⟨complement F = empty-var o 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 oCL selfbutterket default)⟩
by auto
also have ⟨... = one-dim-iso (I a) *C (F (selfbutterket default) oCL 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)) oCL (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 [[simpproc 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 $[register]: \langle compatible\ F\ G \rangle$
shows $\langle clinear\ (\lambda\psi. (F\ \psi\ \otimes_p\ G\ \varphi)) \rangle$
using *assms bounded-clinear.clinear pure-state-bounded-clinear by blast*

method *pure-state-flatten-nested* =

$(subst\ pure-state-nested, (auto; fail)[\beta])^+$

The following method *pure-state-eq* tries to solve a equality where both sides are of the form $F_1(\psi_1) \otimes_p F_2(\psi_2) \otimes_p \dots \otimes_p F_n(\psi_n)$ 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 $F :: \langle bit\ update \Rightarrow 'c::\{finite,default\}\ update \rangle$
and $G :: \langle bit\ update \Rightarrow 'c\ update \rangle$
assumes $[register]: \langle compatible\ F\ G \rangle$
shows $\langle (F;G)\ CNOT\ o_{CL}\ (G;F)\ CNOT\ o_{CL}\ (F;G)\ CNOT = (F;G)\ swap-ell2 \rangle$

proof –

define Z **where** $\langle Z = complement\ (F;G) \rangle$
then have $[register]: \langle compatible\ Z\ F \rangle \langle compatible\ Z\ G \rangle$
using *assms compatible-complement-pair1 compatible-complement-pair2 compatible-sym by blast+*

have $[simp]: \langle iso-register\ (F;(G;Z)) \rangle$
using *Z-def assms complements-complement-pair complements-def by blast*

have $eq1: \langle ((F;G)\ CNOT\ o_{CL}\ (G;F)\ CNOT\ o_{CL}\ (F;G)\ CNOT) *_V\ (F(ket\ f)\ \otimes_p\ G(ket\ g)\ \otimes_p\ Z(ket\ z))$
 $= (F;G)\ swap-ell2 *_V\ (F(ket\ f)\ \otimes_p\ G(ket\ g)\ \otimes_p\ Z(ket\ z)) \rangle$ **for** $f\ g\ z$

proof –

have $\langle (F(ket\ f)\ \otimes_p\ G(ket\ g)\ \otimes_p\ Z(ket\ z)) = ((F;G)\ (ket\ f\ \otimes_s\ ket\ g)\ \otimes_p\ Z(ket\ z)) \rangle$

by *pure-state-eq*

also have $\langle (F;G)\ CNOT *_V \dots = ((F;G)\ (ket\ f\ \otimes_s\ ket\ (g+f))\ \otimes_p\ Z(ket\ z)) \rangle$

apply $(subst\ state-apply1)$ **by** *auto*

also have $\langle \dots = ((G;F)\ (ket\ (g+f)\ \otimes_s\ ket\ f)\ \otimes_p\ Z(ket\ z)) \rangle$

by *pure-state-eq*

also have $\langle (G;F)\ CNOT *_V \dots = ((G;F)\ (ket\ (g+f)\ \otimes_s\ ket\ g)\ \otimes_p\ Z(ket\ z)) \rangle$

apply $(subst\ state-apply1)$ **by** *auto*

also have $\langle \dots = ((F;G)\ (ket\ g\ \otimes_s\ ket\ (g+f))\ \otimes_p\ Z(ket\ z)) \rangle$

by *pure-state-eq*

also have $\langle (F;G)\ CNOT *_V \dots = ((F;G)\ ket\ g\ \otimes_s\ ket\ f\ \otimes_p\ Z(ket\ z)) \rangle$

apply $(subst\ state-apply1)$

apply *simp*

using *add-right-imp-eq by fastforce*

also have $\langle \dots = (F(ket\ g)\ \otimes_p\ G(ket\ f)\ \otimes_p\ Z(ket\ z)) \rangle$

by *pure-state-eq*

finally have $1: \langle ((F;G)\ CNOT\ o_{CL}\ (G;F)\ CNOT\ o_{CL}\ (F;G)\ CNOT) *_V\ (F(ket\ f)\ \otimes_p\ G(ket\ g)\ \otimes_p\ Z(ket\ z)) = (F(ket\ g)\ \otimes_p\ G(ket\ f)\ \otimes_p\ Z(ket\ z)) \rangle$

by *auto*

have $\langle (F(ket\ f)\ \otimes_p\ G(ket\ g)\ \otimes_p\ Z(ket\ z)) = ((F;G)\ (ket\ f\ \otimes_s\ ket\ g)\ \otimes_p\ Z(ket\ z)) \rangle$

by *pure-state-eq*

also have $\langle (F;G)\ swap-ell2 *_V \dots = ((F;G)\ (ket\ g\ \otimes_s\ ket\ f)\ \otimes_p\ Z(ket\ z)) \rangle$

```

    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 mathoverflow 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](https://arxiv.org/abs/2105.10914) [cs.LO], 2021.