

Ribbon Proofs for Separation Logic (Isabelle Formalisation)

John Wickerson

February 6, 2026

Abstract

This document concerns the theory of *ribbon proofs*: a diagrammatic proof system, based on separation logic, for verifying program correctness. We include the syntax, proof rules, and soundness results for two alternative formalisations of ribbon proofs.

Compared to traditional ‘proof outlines’, ribbon proofs emphasise the structure of a proof, so are intelligible and pedagogical. Because they contain less redundancy than proof outlines, and allow each proof step to be checked locally, they may be more scalable. Where proof outlines are cumbersome to modify, ribbon proofs can be visually manoeuvred to yield proofs of variant programs.

Contents

1	Introduction	2
2	Finite partial functions	3
2.1	Difference	3
2.2	Comprehension	3
2.3	Domain	4
2.4	Lookup	4
3	General purpose definitions and lemmas	5
3.1	Projection functions on triples	6
4	Proof chains	6
4.1	Projections	7
4.2	Chain length	7
4.3	Extracting triples from chains	7
4.4	Evaluating a predicate on each triple of a chain	8
4.5	A map function for proof chains	9
4.6	Extending a chain on its right-hand side	10

5	Assertions, commands, and separation logic proof rules	10
5.1	Assertions	11
5.2	Commands	11
5.3	Separation logic proof rules	12
6	Ribbon proof interfaces	13
6.1	Syntax of interfaces	13
6.2	An iterated horizontal-composition operator	15
6.3	Semantics of interfaces	16
6.4	Program variables mentioned in an interface.	16
7	Syntax and proof rules for stratified diagrams	17
7.1	Syntax of stratified diagrams	17
7.2	Proof rules for stratified diagrams	19
7.3	Soundness	19
8	Syntax and proof rules for graphical diagrams	20
8.1	Syntax of graphical diagrams	20
8.2	Well formedness of graphical diagrams	21
8.3	Initial and terminal nodes	22
8.4	Top and bottom interfaces	22
8.5	Proof rules for graphical diagrams	23
8.6	Extracting commands from diagrams	23
9	Soundness for graphical diagrams	25
9.1	Proofstate chains	25
9.2	Interface chains	42
9.3	Soundness proof	43

1 Introduction

Ribbon proofs are a diagrammatic approach for proving program correctness, based on separation logic. They are due to Wickerson, Dodds and Parkinson [4], and are also described in Wickerson’s PhD dissertation [3]. An early version of the proof system, for proving entailments between quantifier-free separation logic assertions, was introduced by Bean [1].

Compared to traditional ‘proof outlines’, ribbon proofs emphasise the structure of a proof, so are intelligible and pedagogical. Because they contain less redundancy than proof outlines, and allow each proof step to be checked locally, they may be more scalable. Where proof outlines are cumbersome to modify, ribbon proofs can be visually manoeuvred to yield proofs of variant programs.

In this document, we formalise a two-dimensional graphical syntax for ribbon proofs, provide proof rules, and show that any provable ribbon proof can be recreated using the ordinary rules of separation logic.

In fact, we provide two different formalisations. Our “stratified” formalisation sees a ribbon proof as a sequence of rows, with each row containing one step of the proof. This formalisation is very simple, but it does not reflect the visual intuition of ribbon proofs, which suggests that some proof steps can be slid up or down without affecting the validity of the overall proof. Our “graphical” formalisation sees a ribbon proof as a graph; specifically, as a directed acyclic nested graph. Ribbon proofs formalised in this way are more manoeuvrable, but proving soundness is trickier, and requires the assumption that separation logic’s Frame rule has no side-condition (an assumption that can be validated by using, for instance, variables-as-resource [2]).

2 Finite partial functions

```
theory More-Finite-Map imports
  HOL-Library.Finite-Map
begin
```

```
lemma fdisjoint-iff:  $A \mid\cap\mid B = \{\mid\}$   $\longleftrightarrow (\forall x. x \mid\in\mid A \longrightarrow x \mid\notin\mid B)$ 
  by (smt (verit) disjoint-iff-fnot-equal)
```

```
unbundle lifting-syntax
unbundle fmap.lifting
```

```
type-notation fmap (infix  $\langle\rightarrow_f\rangle$  9)
```

2.1 Difference

```
definition
  map-diff ::  $('k \rightarrow 'v) \Rightarrow 'k \text{ fset} \Rightarrow ('k \rightarrow 'v)$ 
where
  map-diff f ks = restrict-map f (- fset ks)
```

```
lift-definition
  fmap-diff ::  $('k \rightarrow_f 'v) \Rightarrow 'k \text{ fset} \Rightarrow ('k \rightarrow_f 'v)$  (infix  $\langle\ominus\rangle$  110)
is map-diff
by (auto simp add: map-diff-def)
```

2.2 Comprehension

```
definition
  make-map ::  $'k \text{ fset} \Rightarrow 'v \Rightarrow ('k \rightarrow 'v)$ 
where
  make-map ks v  $\equiv \lambda k. \text{if } k \in \text{fset } ks \text{ then } \text{Some } v \text{ else } \text{None}$ 
```

lemma *make-map-transfer*[*transfer-rule*]: $(rel\text{-}fset (=) ==> A ==> rel\text{-}map A)$ *make-map make-map*
unfolding *make-map-def*
by *transfer-prover*

lemma *dom-make-map*:
 $dom (make\text{-}map\ ks\ v) = fset\ ks$
by (*metis domIff make-map-def not-Some-eq set-eqI*)

lift-definition
 $make\text{-}fmap :: 'k\ fset \Rightarrow 'v \Rightarrow ('k \rightarrow_f 'v) (\langle [- | => -] \rangle)$
is *make-map parametric make-map-transfer*
by (*unfold make-map-def dom-def, auto*)

lemma *make-fmap-empty*[*simp*]: $[\{\}] | => f = fempty$
by *transfer (simp add: make-map-def)*

2.3 Domain

lemma *fmap-add-commute*:
assumes $fmdom\ A\ |\cap|\ fmdom\ B = \{\}$
shows $A\ ++_f\ B = B\ ++_f\ A$
using *assms including fset.lifting*
apply (*transfer*)
apply (*rule ext*)
apply (*auto simp: dom-def map-add-def split: option.splits*)
done

lemma *make-fmap-union*:
 $[xs\ | => v]\ ++_f\ [ys\ | => v] = [xs\ |\cup|\ ys\ | => v]$
by (*transfer, auto simp add: make-map-def map-add-def*)

lemma *fdom-make-fmap*: $fmdom\ [ks\ | => v] = ks$

apply (*subst fmdom-def*)
apply *transfer*
apply (*auto simp: dom-def make-map-def fset-inverse*)
done

2.4 Lookup

lift-definition
 $lookup :: ('k \rightarrow_f 'v) \Rightarrow 'k \Rightarrow 'v$
is (\circ) *the* .

lemma *lookup-make-fmap*:
assumes $k \in fset\ ks$
shows $lookup\ [ks\ | => v]\ k = v$
using *assms by (transfer, simp add: make-map-def)*

lemma *lookup-make-fmap1*:
 $lookup \ [\{|k|\} \ |=> v] \ k = v$
by (*metis finsert.rep-eq insert-iff lookup-make-fmap*)

lemma *lookup-union1*:
assumes $k \in | \ fdom \ ys$
shows $lookup \ (xs \ ++_f \ ys) \ k = lookup \ ys \ k$
using *assms including fset.lifting*
by *transfer auto*

lemma *lookup-union2*:
assumes $k \notin | \ fdom \ ys$
shows $lookup \ (xs \ ++_f \ ys) \ k = lookup \ xs \ k$
using *assms including fset.lifting*
by *transfer (auto simp: map-add-def split: option.splits)*

lemma *lookup-union3*:
assumes $k \notin | \ fdom \ xs$
shows $lookup \ (xs \ ++_f \ ys) \ k = lookup \ ys \ k$
using *assms including fset.lifting*
by *transfer (auto simp: map-add-def split: option.splits)*

end

3 General purpose definitions and lemmas

theory *JHelper* **imports**

Main

begin

lemma *Collect-iff*:
 $a \in \{x \ . \ P \ x\} \equiv P \ a$
by *auto*

lemma *diff-diff-eq*:
assumes $C \subseteq B$
shows $(A - C) - (B - C) = A - B$
using *assms by auto*

lemma *nth-in-set*:
 $\llbracket i < length \ xs \ ; \ xs \ ! \ i = x \rrbracket \implies x \in set \ xs$ **by** *auto*

lemma *disjI [intro]*:
assumes $\neg P \implies Q$
shows $P \vee Q$
using *assms by auto*

lemma *empty-eq-Plus-conv*:
 $(\{\} = A \ <+> B) = (A = \{\} \wedge B = \{\})$

For example, $\{ a \} \cdot \text{proof} \cdot \{ \text{chain} \} \cdot \text{might} \cdot \{ \text{look} \} \cdot \text{like} \cdot \{ \text{this} \}$.

4.1 Projections

Project first assertion.

```
fun
  pre :: ('a,'c) chain  $\Rightarrow$  'a
where
  pre  $\{ P \} = P$ 
| pre ( $\{ P \} \cdot \dots$ ) = P
```

Project final assertion.

```
fun
  post :: ('a,'c) chain  $\Rightarrow$  'a
where
  post  $\{ P \} = P$ 
| post ( $\{ - \} \cdot \dots \Pi$ ) = post  $\Pi$ 
```

Project list of commands.

```
fun
  comlist :: ('a,'c) chain  $\Rightarrow$  'c list
where
  comlist  $\{ - \} = []$ 
| comlist ( $\{ - \} \cdot x \cdot \Pi$ ) = x # (comlist  $\Pi$ )
```

4.2 Chain length

```
fun
  chainlen :: ('a,'c) chain  $\Rightarrow$  nat
where
  chainlen  $\{ - \} = 0$ 
| chainlen ( $\{ - \} \cdot \dots \Pi$ ) = 1 + (chainlen  $\Pi$ )
```

lemma *len-comlist-chainlen*:

```
length (comlist  $\Pi$ ) = chainlen  $\Pi$ 
by (induct  $\Pi$ , auto)
```

4.3 Extracting triples from chains

nthtriple Π n extracts the n th triple of Π , counting from 0. The function is well-defined when $n < \text{chainlen } \Pi$.

```
fun
  nthtriple :: ('a,'c) chain  $\Rightarrow$  nat  $\Rightarrow$  ('a * 'c * 'a)
where
  nthtriple ( $\{ P \} \cdot x \cdot \Pi$ ) 0 = (P, x, pre  $\Pi$ )
| nthtriple ( $\{ P \} \cdot x \cdot \Pi$ ) (Suc n) = nthtriple  $\Pi$  n
```

The list of middle components of Π 's triples is the list of Π 's commands.

```

lemma snds-of-triples-form-comlist:
  fixes  $\Pi$   $i$ 
  shows  $i < \text{chainlen } \Pi \implies \text{snd3 } (\text{nthtriple } \Pi \ i) = (\text{comlist } \Pi)!i$ 
apply (induct  $\Pi$  arbitrary:  $i$ )
apply simp
apply (case-tac  $i$ )
apply (auto simp add: snd3-def)
done

```

4.4 Evaluating a predicate on each triple of a chain

chain-all φ holds of Π iff φ holds for each of Π 's individual triples.

```

fun
  chain-all ::  $((\ 'a \times \ 'c \times \ 'a) \Rightarrow \text{bool}) \Rightarrow (\ 'a, \ 'c) \text{ chain} \Rightarrow \text{bool}$ 
where
  chain-all  $\varphi \ \{\!| \ \sigma \ |\!| = \text{True}$ 
| chain-all  $\varphi \ (\{\!| \ \sigma \ |\!| \cdot x \cdot \Pi) = (\varphi \ (\sigma, x, \text{pre } \Pi) \wedge \text{chain-all } \varphi \ \Pi)$ 

```

```

lemma chain-all-mono [mono]:
   $x \leq y \implies \text{chain-all } x \leq \text{chain-all } y$ 
proof (intro le-funI le-boolI)
  fix  $f \ g :: (\ 'a \times \ 'b \times \ 'a) \Rightarrow \text{bool}$ 
  fix  $\Pi :: (\ 'a, \ 'b) \text{ chain}$ 
  assume  $f \leq g$ 
  assume chain-all  $f \ \Pi$ 
  thus chain-all  $g \ \Pi$ 
  apply (induct  $\Pi$ )
  apply simp
  apply (metis  $\langle f \leq g \rangle \text{chain-all.simps}(2) \text{predicate1D}$ )
  done
qed

```

```

lemma chain-all-nthtriple:
   $(\text{chain-all } \varphi \ \Pi) = (\forall i < \text{chainlen } \Pi. \varphi \ (\text{nthtriple } \Pi \ i))$ 
proof (intro iffI allI impI)
  fix  $i$ 
  assume chain-all  $\varphi \ \Pi$  and  $i < \text{chainlen } \Pi$ 
  thus  $\varphi \ (\text{nthtriple } \Pi \ i)$ 
  proof (induct i arbitrary:  $\Pi$ )
    case  $0$ 
    then obtain  $\sigma \ x \ \Pi'$  where  $\Pi\text{-def}: \Pi = \{\!| \ \sigma \ |\!| \cdot x \cdot \Pi'$ 
    by (metis chain.exhaust chainlen.simps(1) less-nat-zero-code)
    show ?case
    by (insert 0.premis(1), unfold  $\Pi\text{-def}$ , auto)
  next
  case (Suc  $i$ )
  then obtain  $\sigma \ x \ \Pi'$  where  $\Pi\text{-def}: \Pi = \{\!| \ \sigma \ |\!| \cdot x \cdot \Pi'$ 
  by (metis chain.exhaust chainlen.simps(1) less-nat-zero-code)
  show ?case

```

```

  apply (unfold  $\Pi$ -def nthtriple.simps)
  apply (intro Suc.hyps, insert Suc.prem, auto simp add:  $\Pi$ -def)
done
qed
next
assume  $\forall i < \text{chainlen } \Pi. \varphi (\text{nthtriple } \Pi i)$ 
hence  $\bigwedge i. i < \text{chainlen } \Pi \implies \varphi (\text{nthtriple } \Pi i)$  by auto
thus chain-all  $\varphi \Pi$ 
proof (induct  $\Pi$ )
  case (cCons  $\sigma x \Pi'$ )
  show ?case
  apply (simp, intro conjI)
  apply (subgoal-tac  $\varphi (\text{nthtriple } (\{\sigma\} \cdot x \cdot \Pi') 0)$ , simp)
  apply (intro cCons.prem, simp)
  apply (intro cCons.hyps)
  proof -
    fix  $i$ 
    assume  $i < \text{chainlen } \Pi'$ 
    hence  $\text{Suc } i < \text{chainlen } (\{\sigma\} \cdot x \cdot \Pi')$  by auto
    from cCons.prem[OF this] show  $\varphi (\text{nthtriple } \Pi' i)$  by auto
  qed
qed(auto)
qed

```

4.5 A map function for proof chains

$\text{chainmap } f g \Pi$ maps f over each of Π 's assertions, and g over each of Π 's commands.

```

fun
  chainmap :: ('a  $\Rightarrow$  'b)  $\Rightarrow$  ('c  $\Rightarrow$  'd)  $\Rightarrow$  ('a,'c) chain  $\Rightarrow$  ('b,'d) chain
where
  chainmap f g  $\{\{P\}\} = \{\{f P\}\}$ 
| chainmap f g ( $\{\{P\}\} \cdot x \cdot \Pi$ ) =  $\{\{f P\}\} \cdot g x \cdot \text{chainmap } f g \Pi$ 

```

Mapping over a chain preserves its length.

lemma *chainmap-preserves-length*:
 $\text{chainlen } (\text{chainmap } f g \Pi) = \text{chainlen } \Pi$
by (induct Π , auto)

lemma *pre-chainmap*:
 $\text{pre } (\text{chainmap } f g \Pi) = f (\text{pre } \Pi)$
by (induct Π , auto)

lemma *post-chainmap*:
 $\text{post } (\text{chainmap } f g \Pi) = f (\text{post } \Pi)$
by (induct Π , auto)

lemma *nthtriple-chainmap*:

```

assumes  $i < \text{chainlen } \Pi$ 
shows  $\text{nthtriple } (\text{chainmap } f g \Pi) i$ 
  =  $(\lambda t. (f (\text{fst3 } t), g (\text{snd3 } t), f (\text{thd3 } t))) (\text{nthtriple } \Pi i)$ 
using assms
proof (induct  $\Pi$  arbitrary: i)
  case cCons
  thus ?case
  by (induct i, auto simp add: pre-chainmap fst3-def snd3-def thd3-def)
qed (auto)

```

4.6 Extending a chain on its right-hand side

```

fun
   $cSnoc :: ('a, 'c) \text{chain} \Rightarrow 'c \Rightarrow 'a \Rightarrow ('a, 'c) \text{chain}$ 
where
   $cSnoc \{\!\! \{ \sigma \}\!\! \} y \tau = \{\!\! \{ \sigma \}\!\! \} \cdot y \cdot \{\!\! \{ \tau \}\!\! \}$ 
|  $cSnoc (\{\!\! \{ \sigma \}\!\! \} \cdot x \cdot \Pi) y \tau = \{\!\! \{ \sigma \}\!\! \} \cdot x \cdot (cSnoc \Pi y \tau)$ 

```

```

lemma len-snoc:
  fixes  $\Pi x P$ 
  shows  $\text{chainlen } (cSnoc \Pi x P) = 1 + (\text{chainlen } \Pi)$ 
by (induct  $\Pi, auto$ )

```

```

lemma pre-snoc:
   $\text{pre } (cSnoc \Pi x P) = \text{pre } \Pi$ 
by (induct  $\Pi, auto$ )

```

```

lemma post-snoc:
   $\text{post } (cSnoc \Pi x P) = P$ 
by (induct  $\Pi, auto$ )

```

```

lemma comlist-snoc:
   $\text{comlist } (cSnoc \Pi x p) = \text{comlist } \Pi @ [x]$ 
by (induct  $\Pi, auto$ )

```

end

5 Assertions, commands, and separation logic proof rules

```

theory Ribbons-Basic imports
  Main
begin

```

We define a command language, assertions, and the rules of separation logic, plus some derived rules that are used by our tool. This is the only theory

file that is loaded by the tool. We keep it as small as possible.

5.1 Assertions

The language of assertions includes (at least) an emp constant, a star-operator, and existentially-quantified logical variables.

typedecl *assertion*

axiomatization

Emp :: *assertion*

axiomatization

Star :: *assertion* \Rightarrow *assertion* \Rightarrow *assertion* (**infixr** $\langle \star \rangle$ 55)

where

star-comm: $p \star q = q \star p$ **and**

star-assoc: $(p \star q) \star r = p \star (q \star r)$ **and**

star-emp: $p \star \text{Emp} = p$ **and**

emp-star: $\text{Emp} \star p = p$

lemma *star-rot*:

$q \star p \star r = p \star q \star r$

using *star-assoc star-comm* **by** *auto*

axiomatization

Exists :: *string* \Rightarrow *assertion* \Rightarrow *assertion*

Extracting the set of program variables mentioned in an assertion.

axiomatization

rd-ass :: *assertion* \Rightarrow *string set*

where *rd-emp*: *rd-ass* *Emp* = {}

and *rd-star*: *rd-ass* $(p \star q) = \text{rd-ass } p \cup \text{rd-ass } q$

and *rd-exists*: *rd-ass* $(\text{Exists } x p) = \text{rd-ass } p$

5.2 Commands

The language of commands comprises (at least) non-deterministic choice, non-deterministic looping, skip and sequencing.

typedecl *command*

axiomatization

Choose :: *command* \Rightarrow *command* \Rightarrow *command*

axiomatization

Loop :: *command* \Rightarrow *command*

axiomatization

Skip :: *command*

axiomatization

```
Seq :: command ⇒ command ⇒ command (infixr <;> 55)
where seq-assoc: c1 ;; (c2 ;; c3) = (c1 ;; c2) ;; c3
and seq-skip: c ;; Skip = c
and skip-seq: Skip ;; c = c
```

Extracting the set of program variables read by a command.

axiomatization

```
rd-com :: command ⇒ string set
where rd-com-choose: rd-com (Choose c1 c2) = rd-com c1 ∪ rd-com c2
and rd-com-loop: rd-com (Loop c) = rd-com c
and rd-com-skip: rd-com (Skip) = {}
and rd-com-seq: rd-com (c1 ;; c2) = rd-com c1 ∪ rd-com c2
```

Extracting the set of program variables written by a command.

axiomatization

```
wr-com :: command ⇒ string set
where wr-com-choose: wr-com (Choose c1 c2) = wr-com c1 ∪ wr-com c2
and wr-com-loop: wr-com (Loop c) = wr-com c
and wr-com-skip: wr-com (Skip) = {}
and wr-com-seq: wr-com (c1 ;; c2) = wr-com c1 ∪ wr-com c2
```

5.3 Separation logic proof rules

Note that the frame rule has a side-condition concerning program variables. When proving the soundness of our graphical formalisation of ribbon proofs, we shall omit this side-condition.

inductive

```
prov-triple :: assertion × command × assertion ⇒ bool
where
  exists: prov-triple (p, c, q) ⇒⇒ prov-triple (Exists x p, c, Exists x q)
  | choose: [ prov-triple (p, c1, q); prov-triple (p, c2, q) ]
    ⇒⇒ prov-triple (p, Choose c1 c2, q)
  | loop: prov-triple (p, c, p) ⇒⇒ prov-triple (p, Loop c, p)
  | frame: [ prov-triple (p, c, q); wr-com(c) ∩ rd-ass(r) = {} ]
    ⇒⇒ prov-triple (p ★ r, c, q ★ r)
  | skip: prov-triple (p, Skip, p)
  | seq: [ prov-triple (p, c1, q); prov-triple (q, c2, r) ]
    ⇒⇒ prov-triple (p, c1 ;; c2, r)
```

Here are some derived proof rules, which are used in our ribbon-checking tool.

lemma choice-lemma:

```
assumes prov-triple (p1, c1, q1) and prov-triple (p2, c2, q2)
and p = p1 and p1 = p2 and q = q1 and q1 = q2
shows prov-triple (p, Choose c1 c2, q)
using assms prov-triple.choose by auto
```

```

lemma loop-lemma:
  assumes prov-triple (p1, c, q1) and p = p1 and p1 = q1 and q1 = q
  shows prov-triple (p, Loop c, q)
using assms prov-triple.loop by auto

```

```

lemma seq-lemma:
  assumes prov-triple (p1, c1, q1) and prov-triple (p2, c2, q2)
  and q1 = p2
  shows prov-triple (p1, c1 ;; c2, q2)
using assms prov-triple.seq by auto

```

end

6 Ribbon proof interfaces

theory *Ribbons-Interfaces* **imports**

Ribbons-Basic

Proofchain

HOL-Library.FSet

begin

Interfaces are the top and bottom boundaries through which diagrams can be connected into a surrounding context. For instance, when composing two diagrams vertically, the bottom interface of the upper diagram must match the top interface of the lower diagram.

We define a datatype of concrete interfaces. We then quotient by the associativity, commutativity and unity properties of our horizontal-composition operator.

6.1 Syntax of interfaces

```

datatype conc-interface =
  Ribbon-conc assertion
| HComp-int-conc conc-interface conc-interface (infix  $\langle \otimes_c \rangle$  50)
| Emp-int-conc ( $\langle \varepsilon_c \rangle$ )
| Exists-int-conc string conc-interface

```

We define an equivalence on interfaces. The first three rules make this an equivalence relation. The next three make it a congruence. The next two identify interfaces up to associativity and commutativity of (\otimes_c) . The final two make ε_c the left and right unit of (\otimes_c) .

inductive

equiv-int :: *conc-interface* \Rightarrow *conc-interface* \Rightarrow *bool* (**infix** $\langle \simeq \rangle$ 45)

where

refl: $P \simeq P$

| *sym*: $P \simeq Q \Longrightarrow Q \simeq P$

```

| trans:  $[P \simeq Q; Q \simeq R] \implies P \simeq R$ 
| cong-hcomp1:  $P \simeq Q \implies P' \otimes_c P \simeq P' \otimes_c Q$ 
| cong-hcomp2:  $P \simeq Q \implies P \otimes_c P' \simeq Q \otimes_c P'$ 
| cong-exists:  $P \simeq Q \implies \text{Exists-int-conc } x P \simeq \text{Exists-int-conc } x Q$ 
| hcomp-conc-assoc:  $P \otimes_c (Q \otimes_c R) \simeq (P \otimes_c Q) \otimes_c R$ 
| hcomp-conc-comm:  $P \otimes_c Q \simeq Q \otimes_c P$ 
| hcomp-conc-unit1:  $\varepsilon_c \otimes_c P \simeq P$ 
| hcomp-conc-unit2:  $P \otimes_c \varepsilon_c \simeq P$ 

```

lemma *equiv-int-cong-hcomp*:

```

 $[P \simeq Q; P' \simeq Q'] \implies P \otimes_c P' \simeq Q \otimes_c Q'$ 
by (metis cong-hcomp1 cong-hcomp2 equiv-int.trans)

```

quotient-type *interface* = *conc-interface* / *equiv-int*

```

apply (intro equivI)
apply (intro reflI, simp add: equiv-int.refl)
apply (intro sympI, simp add: equiv-int.sym)
apply (intro transI, elim equiv-int.trans, simp add: equiv-int.refl)
done

```

lift-definition

```

Ribbon :: assertion  $\Rightarrow$  interface
is Ribbon-conc .

```

lift-definition

```

Emp-int :: interface  $(\langle \varepsilon \rangle)$ 
is  $\varepsilon_c$  .

```

lift-definition

```

Exists-int :: string  $\Rightarrow$  interface  $\Rightarrow$  interface
is Exists-int-conc
by (rule equiv-int.cong-exists)

```

lift-definition

```

HComp-int :: interface  $\Rightarrow$  interface  $\Rightarrow$  interface (infix  $\langle \otimes \rangle$  50)
is HComp-int-conc by (rule equiv-int-cong-hcomp)

```

lemma *hcomp-comm*:

```

 $(P \otimes Q) = (Q \otimes P)$ 
by (rule hcomp-conc-comm[Transfer.transferred])

```

lemma *hcomp-assoc*:

```

 $(P \otimes (Q \otimes R)) = ((P \otimes Q) \otimes R)$ 
by (rule hcomp-conc-assoc[Transfer.transferred])

```

lemma *emp-hcomp*:

```

 $\varepsilon \otimes P = P$ 
by (rule hcomp-conc-unit1[Transfer.transferred])

```

lemma *hcomp-emp*:

$$P \otimes \varepsilon = P$$

by (rule *hcomp-conc-unit2*[*Transfer.transferred*])

lemma *comp-fun-commute-hcomp*:

$$\text{comp-fun-commute } (\otimes)$$

by *standard* (simp add: *hcomp-assoc fun-eq-iff, metis hcomp-comm*)

6.2 An iterated horizontal-composition operator

definition *iter-hcomp* :: ('a fset) \Rightarrow ('a \Rightarrow interface) \Rightarrow interface

where

$$\text{iter-hcomp } X f \equiv \text{ffold } ((\otimes) \circ f) \varepsilon X$$

syntax *iter-hcomp-syntax* ::

$$'a \Rightarrow ('a \text{ fset}) \Rightarrow ('a \Rightarrow \text{interface}) \Rightarrow \text{interface}$$

$$\langle\langle (\otimes) \text{ -} | \in | \text{ -} \rangle \rangle [0, 0, 10] 10$$

syntax-consts *iter-hcomp-syntax* == *iter-hcomp*

translations $\otimes x | \in | M. e == \text{CONST } \text{iter-hcomp } M (\lambda x. e)$

term $\otimes P | \in | Ps. f P$ — this is eta-expanded, so prints in expanded form

term $\otimes P | \in | Ps. f$ — this isn't eta-expanded, so prints as written

lemma *iter-hcomp-cong*:

$$\text{assumes } \forall v \in \text{fset } vs. \varphi v = \varphi' v$$

$$\text{shows } (\otimes v | \in | vs. \varphi v) = (\otimes v | \in | vs. \varphi' v)$$

using *assms unfolding iter-hcomp-def*

by (auto simp add: *comp-fun-commute.comp-comp-fun-commute comp-fun-commute-hcomp*

intro: ffold-cong)

lemma *iter-hcomp-empty*:

$$\text{shows } (\otimes x | \in | \{\}\}. p x) = \varepsilon$$

by (simp add: *comp-fun-commute.comp-comp-fun-commute comp-fun-commute.ffold-empty comp-fun-commute-hcomp iter-hcomp-def*)

lemma *iter-hcomp-insert*:

$$\text{assumes } v \notin ws$$

$$\text{shows } (\otimes x | \in | \text{finsert } v \text{ } ws. p x) = (p v \otimes (\otimes x | \in | ws. p x))$$

proof —

$$\text{interpret } \text{comp-fun-commute } ((\otimes) \circ p)$$

by (*metis comp-fun-commute.comp-comp-fun-commute comp-fun-commute-hcomp*)

from *assms show ?thesis unfolding iter-hcomp-def by auto*

qed

lemma *iter-hcomp-union*:

$$\text{assumes } vs \cap ws = \{\}$$

$$\text{shows } (\otimes x | \in | vs \cup ws. p x) = ((\otimes x | \in | vs. p x) \otimes (\otimes x | \in | ws. p x))$$

using *assms*
by (*induct vs*) (*auto simp add: emp-hcomp iter-hcomp-empty iter-hcomp-insert hcomp-assoc*)

6.3 Semantics of interfaces

The semantics of an interface is an assertion.

fun
conc-asn :: *conc-interface* \Rightarrow *assertion*
where
conc-asn (*Ribbon-conc* *p*) = *p*
| *conc-asn* (*P* \otimes_c *Q*) = (*conc-asn* *P*) \star (*conc-asn* *Q*)
| *conc-asn* (ε_c) = *Emp*
| *conc-asn* (*Exists-int-conc* *x P*) = *Exists* *x* (*conc-asn* *P*)

lift-definition

asn :: *interface* \Rightarrow *assertion*
is *conc-asn*
by (*induct-tac rule:equiv-int.induct*) (*auto simp add: star-assoc star-comm star-rot emp-star*)

lemma *asn-simps* [*simp*]:

asn (*Ribbon* *p*) = *p*
asn (*P* \otimes *Q*) = (*asn* *P*) \star (*asn* *Q*)
asn ε = *Emp*
asn (*Exists-int* *x P*) = *Exists* *x* (*asn* *P*)
by (*transfer, simp*) $+$

6.4 Program variables mentioned in an interface.

fun
rd-conc-int :: *conc-interface* \Rightarrow *string set*
where
rd-conc-int (*Ribbon-conc* *p*) = *rd-ass* *p*
| *rd-conc-int* (*P* \otimes_c *Q*) = *rd-conc-int* *P* \cup *rd-conc-int* *Q*
| *rd-conc-int* (ε_c) = $\{\}$
| *rd-conc-int* (*Exists-int-conc* *x P*) = *rd-conc-int* *P*

lift-definition

rd-int :: *interface* \Rightarrow *string set*
is *rd-conc-int*
by (*induct-tac rule: equiv-int.induct*) *auto*

The program variables read by an interface are the same as those read by its corresponding assertion.

lemma *rd-int-is-rd-ass*:

rd-ass (*asn* *P*) = *rd-int* *P*
by (*transfer, induct-tac P, auto simp add: rd-star rd-exists rd-emp*)

Here is an iterated version of the Hoare logic sequencing rule.

lemma *seq-fold*:

$\bigwedge \Pi. \llbracket \text{length } cs = \text{chainlen } \Pi ; p1 = \text{asn } (\text{pre } \Pi) ; p2 = \text{asn } (\text{post } \Pi) ;$
 $\bigwedge i. i < \text{chainlen } \Pi \implies \text{prov-triple}$
 $(\text{asn } (\text{fst3 } (\text{nthtriple } \Pi i)), cs ! i, \text{asn } (\text{thd3 } (\text{nthtriple } \Pi i))) \rrbracket$
 $\implies \text{prov-triple } (p1, \text{foldr } (;;) cs \text{Skip}, p2)$

proof (*induct cs arbitrary: p1 p2*)

case *Nil*

thus *?case*

by (*cases* Π , *auto simp add: prov-triple.skip*)

next

case (*Cons c cs*)

obtain $p x \Pi'$ **where** $\Pi\text{-def}: \Pi = \{ p \} \cdot x \cdot \Pi'$

by (*metis Cons.prem(1) chain.exhaust chainlen.simps(1) impossible-Cons le0*)

show *?case*

apply (*unfold foldr-Cons o-def*)

apply (*rule prov-triple.seq[where q = asn (pre Π')]*)

apply (*unfold Cons.prem(2) Cons.prem(3) $\Pi\text{-def}$ pre.simps post.simps*)

apply (*subst nth-Cons-0[of c cs, symmetric]*)

apply (*subst fst3-simp[of p x pre Π' , symmetric]*)

apply (*subst(2) thd3-simp[of p x pre Π' , symmetric]*)

apply (*subst(1 2) nthtriple.simps(1)[of p x Π' , symmetric]*)

apply (*fold $\Pi\text{-def}$, intro Cons.prem(4), simp add: $\Pi\text{-def}$*)

apply (*intro Cons.hyps, insert $\Pi\text{-def}$ Cons.prem(1), auto*)

apply (*fold nth-Cons-Suc[of c cs] nthtriple.simps(2)[of p x Π']*)

apply (*fold $\Pi\text{-def}$, intro Cons.prem(4), simp add: $\Pi\text{-def}$*)

done

qed

end

7 Syntax and proof rules for stratified diagrams

theory *Ribbons-Stratified imports*

Ribbons-Interfaces

Proofchain

begin

We define the syntax of stratified diagrams. We give proof rules for stratified diagrams, and prove them sound with respect to the ordinary rules of separation logic.

7.1 Syntax of stratified diagrams

datatype *sdiagram* = *SDiagram* (*cell* \times *interface*) *list*

and *cell* =

Filler interface

| *Basic interface command interface*

```

| Exists-sdia string sdiagram
| Choose-sdia interface sdiagram sdiagram interface
| Loop-sdia interface sdiagram interface

```

datatype-compat *sdiagram cell*

type-synonym *row = cell × interface*

Extracting the command from a stratified diagram.

```

fun
  com-sdia :: sdiagram ⇒ command and
  com-cell :: cell ⇒ command
where
  com-sdia (SDiagram  $\varrho$ s) = foldr (;;) (map (com-cell ∘ fst)  $\varrho$ s) Skip
| com-cell (Filler P) = Skip
| com-cell (Basic P c Q) = c
| com-cell (Exists-sdia x D) = com-sdia D
| com-cell (Choose-sdia P D E Q) = Choose (com-sdia D) (com-sdia E)
| com-cell (Loop-sdia P D Q) = Loop (com-sdia D)

```

Extracting the program variables written by a stratified diagram.

```

fun
  wr-sdia :: sdiagram ⇒ string set and
  wr-cell :: cell ⇒ string set
where
  wr-sdia (SDiagram  $\varrho$ s) = (∪ r ∈ set  $\varrho$ s. wr-cell (fst r))
| wr-cell (Filler P) = {}
| wr-cell (Basic P c Q) = wr-com c
| wr-cell (Exists-sdia x D) = wr-sdia D
| wr-cell (Choose-sdia P D E Q) = wr-sdia D ∪ wr-sdia E
| wr-cell (Loop-sdia P D Q) = wr-sdia D

```

The program variables written by a stratified diagram correspond to those written by the commands therein.

```

lemma wr-sdia-is-wr-com:
  fixes  $\varrho$ s :: row list
  and  $\varrho$  :: row
  shows (wr-sdia D = wr-com (com-sdia D))
  and (wr-cell  $\gamma$  = wr-com (com-cell  $\gamma$ ))
  and (∪  $\varrho$  ∈ set  $\varrho$ s. wr-cell (fst  $\varrho$ ))
    = wr-com (foldr (;;) (map ( $\lambda(\gamma, F)$ . com-cell  $\gamma$ )  $\varrho$ s) Skip)
  and wr-cell (fst  $\varrho$ ) = wr-com (com-cell (fst  $\varrho$ ))
apply (induct D and  $\gamma$  and  $\varrho$ s and  $\varrho$  rule: compat-sdiagram.induct compat-cell.induct
  compat-cell-interface-prod-list.induct compat-cell-interface-prod.induct)
apply (auto simp add: wr-com-skip wr-com-choose
  wr-com-loop wr-com-seq split-def o-def)
done

```

7.2 Proof rules for stratified diagrams

inductive

$prov\text{-}sdia :: [sdiagram, interface, interface] \Rightarrow bool$ **and**
 $prov\text{-}row :: [row, interface, interface] \Rightarrow bool$ **and**
 $prov\text{-}cell :: [cell, interface, interface] \Rightarrow bool$

where

$SRibbon: prov\text{-}cell (Filler P) P P$
 $| SBasic: prov\text{-}triple (asn P, c, asn Q) \Longrightarrow prov\text{-}cell (Basic P c Q) P Q$
 $| SExists: prov\text{-}sdia D P Q$
 $\quad \Longrightarrow prov\text{-}cell (Exists\text{-}sdia x D) (Exists\text{-}int x P) (Exists\text{-}int x Q)$
 $| SChoice: \llbracket prov\text{-}sdia D P Q ; prov\text{-}sdia E P Q \rrbracket$
 $\quad \Longrightarrow prov\text{-}cell (Choose\text{-}sdia P D E Q) P Q$
 $| SLoop: prov\text{-}sdia D P P \Longrightarrow prov\text{-}cell (Loop\text{-}sdia P D P) P P$
 $| SRow: \llbracket prov\text{-}cell \gamma P Q ; wr\text{-}cell \gamma \cap rd\text{-}int F = \{\} \rrbracket$
 $\quad \Longrightarrow prov\text{-}row (\gamma, F) (P \otimes F) (Q \otimes F)$
 $| SMain: \llbracket chain\text{-}all (\lambda(P,\varrho,Q). prov\text{-}row \varrho P Q) \Pi ; 0 < chainlen \Pi \rrbracket$
 $\quad \Longrightarrow prov\text{-}sdia (SDiagram (comlist \Pi)) (pre \Pi) (post \Pi)$

7.3 Soundness

lemma *soundness-strat-helper:*

$(prov\text{-}sdia D P Q \longrightarrow prov\text{-}triple (asn P, com\text{-}sdia D, asn Q)) \wedge$
 $(prov\text{-}row \varrho P Q \longrightarrow prov\text{-}triple (asn P, com\text{-}cell (fst \varrho), asn Q)) \wedge$
 $(prov\text{-}cell \gamma P Q \longrightarrow prov\text{-}triple (asn P, com\text{-}cell \gamma, asn Q))$

proof (*induct rule: prov-sdia-prov-row-prov-cell.induct*)

case ($SRibbon P$)

show $?case$ **by** (*auto simp add: prov-triple.skip*)

next

case ($SBasic P c Q$)

thus $?case$ **by** *auto*

next

case ($SExists D P Q x$)

thus $?case$ **by** (*auto simp add: prov-triple.exists*)

next

case ($SChoice D P Q E$)

thus $?case$ **by** (*auto simp add: prov-triple.choose*)

next

case ($SLoop D P$)

thus $?case$ **by** (*auto simp add: prov-triple.loop*)

next

case ($SRow \gamma P Q F$)

thus $?case$

by (*simp add: prov-triple.frame rd-int-is-rd-ass wr-sdia-is-wr-com(2)*)

next

case ($SMain \Pi$)

thus $?case$

apply (*unfold com-sdia.simps*)

apply (*intro seq-fold[of - \Pi]*)

apply (*simp-all add: len-comlist-chainlen*)[3]

```

apply (induct  $\Pi$ , simp)
apply (case-tac i, auto simp add: fst3-simp thd3-simp)
done
qed

```

```

corollary soundness-strat:
  assumes prov-sdia D P Q
  shows prov-triple (asn P, com-sdia D, asn Q)
using assms soundness-strat-helper by auto

end

```

8 Syntax and proof rules for graphical diagrams

```

theory Ribbons-Graphical imports
  Ribbons-Interfaces
begin

```

We introduce a graphical syntax for diagrams, describe how to extract commands and interfaces, and give proof rules for graphical diagrams.

8.1 Syntax of graphical diagrams

Fix a type for node identifiers

```

typedecl node

```

Note that this datatype is necessarily an overapproximation of syntactically-wellformed diagrams, for the reason that we can't impose the well-formedness constraints while maintaining admissibility of the datatype declarations. So, we shall impose well-formedness in a separate definition.

```

datatype assertion-gadget =
  Rib assertion
| Exists-dia string diagram
and command-gadget =
  Com command
| Choose-dia diagram diagram
| Loop-dia diagram
and diagram = Graph
  node fset
  node  $\Rightarrow$  assertion-gadget
  (node fset  $\times$  command-gadget  $\times$  node fset) list
type-synonym labelling = node  $\Rightarrow$  assertion-gadget
type-synonym edge = node fset  $\times$  command-gadget  $\times$  node fset

```

Projecting components from a graph

```

fun vertices :: diagram  $\Rightarrow$  node fset ( $\langle \cdot \hat{\ } V \rangle$  [1000] 1000)
where (Graph V  $\wedge$  E)  $\hat{\ } V = V$ 

```

term $this (is \wedge V) = (a \text{ test}) \wedge V$

fun $labelling :: diagram \Rightarrow labelling (\langle \cdot \wedge \Lambda \rangle [1000] 1000)$
where $(Graph \ V \ \Lambda \ E) \wedge \Lambda = \Lambda$

fun $edges :: diagram \Rightarrow edge \ list (\langle \cdot \wedge E \rangle [1000] 1000)$
where $(Graph \ V \ \Lambda \ E) \wedge E = E$

8.2 Well formedness of graphical diagrams

definition $acyclicity :: edge \ list \Rightarrow bool$

where

$acyclicity \ E \equiv acyclic (\bigcup e \in set \ E. fset (fst3 \ e) \times fset (thd3 \ e))$

definition $linearity :: edge \ list \Rightarrow bool$

where

$linearity \ E \equiv$
 $distinct \ E \wedge (\forall e \in set \ E. \forall f \in set \ E. e \neq f \longrightarrow$
 $fst3 \ e \ | \cap | \ fst3 \ f = \{\}\} \wedge$
 $thd3 \ e \ | \cap | \ thd3 \ f = \{\}\})$

lemma $linearityD$:

assumes $linearity \ E$

shows $distinct \ E$

and $\bigwedge e \ f. \llbracket e \in set \ E ; f \in set \ E ; e \neq f \rrbracket \Longrightarrow$

$fst3 \ e \ | \cap | \ fst3 \ f = \{\}\} \wedge$

$thd3 \ e \ | \cap | \ thd3 \ f = \{\}\}$

using $assms \ unfolding \ linearity-def \ by \ auto$

lemma $linearityD2$:

$linearity \ E \Longrightarrow (\forall e \ f. e \in set \ E \wedge f \in set \ E \wedge e \neq f \longrightarrow$

$fst3 \ e \ | \cap | \ fst3 \ f = \{\}\} \wedge$

$thd3 \ e \ | \cap | \ thd3 \ f = \{\}\})$

unfolding $linearity-def \ by \ auto$

inductive

$wf-ass :: assertion-gadget \Rightarrow bool$ **and**

$wf-com :: command-gadget \Rightarrow bool$ **and**

$wf-dia :: diagram \Rightarrow bool$

where

$wf-rib: wf-ass (Rib \ p)$

| $wf-exists: wf-dia \ G \Longrightarrow wf-ass (Exists-dia \ x \ G)$

| $wf-com: wf-com (Com \ c)$

| $wf-choice: \llbracket wf-dia \ G ; wf-dia \ H \rrbracket \Longrightarrow wf-com (Choose-dia \ G \ H)$

| $wf-loop: wf-dia \ G \Longrightarrow wf-com (Loop-dia \ G)$

| $wf-dia: \llbracket \forall e \in set \ E. wf-com (snd3 \ e) ; \forall v \in fset \ V. wf-ass (\Lambda \ v) ;$
 $acyclicity \ E ; linearity \ E ; \forall e \in set \ E. fst3 \ e \ | \cup | \ thd3 \ e \ | \subseteq | \ V \rrbracket \Longrightarrow$
 $wf-dia (Graph \ V \ \Lambda \ E)$

inductive-cases *wf-dia-inv'*: *wf-dia* (*Graph* *V* Λ *E*)

lemma *wf-dia-inv*:

assumes *wf-dia* (*Graph* *V* Λ *E*)

shows $\forall v \in \text{fset } V. \text{wf-ass } (\Lambda v)$

and $\forall e \in \text{set } E. \text{wf-com } (\text{snd3 } e)$

and *acyclicity* *E*

and *linearity* *E*

and $\forall e \in \text{set } E. \text{fst3 } e \mid \cup \mid \text{thd3 } e \mid \subseteq \mid V$

using *assms*

apply –

apply (*elim wf-dia-inv'*, *simp*)+

done

8.3 Initial and terminal nodes

definition

initials :: *diagram* \Rightarrow *node fset*

where

initials *G* = *ffilter* ($\lambda v. (\forall e \in \text{set } G \hat{=} E. v \notin \text{thd3 } e)$) *G* $\hat{=} V$

definition

terminals :: *diagram* \Rightarrow *node fset*

where

terminals *G* = *ffilter* ($\lambda v. (\forall e \in \text{set } G \hat{=} E. v \notin \text{fst3 } e)$) *G* $\hat{=} V$

lemma *no-edges-imp-all-nodes-initial*:

initials (*Graph* *V* Λ []) = *V*

by (*auto simp add: initials-def*)

lemma *no-edges-imp-all-nodes-terminal*:

terminals (*Graph* *V* Λ []) = *V*

by (*auto simp add: terminals-def*)

lemma *initials-in-vertices*:

initials *G* $\mid \subseteq \mid G \hat{=} V$

unfolding *initials-def* **by** *auto*

lemma *terminals-in-vertices*:

terminals *G* $\mid \subseteq \mid G \hat{=} V$

unfolding *terminals-def* **by** *auto*

8.4 Top and bottom interfaces

primrec

top-ass :: *assertion-gadget* \Rightarrow *interface* **and**

top-dia :: *diagram* \Rightarrow *interface*

where

top-dia (*Graph* *V* Λ *E*) = ($\otimes v \mid \in \mid \text{initials } (\text{Graph } V \Lambda E). \text{top-ass } (\Lambda v)$)

| $top\text{-}ass (Rib\ p) = Ribbon\ p$
| $top\text{-}ass (Exists\text{-}dia\ x\ G) = Exists\text{-}int\ x\ (top\text{-}dia\ G)$

primrec

$bot\text{-}ass :: assertion\text{-}gadget \Rightarrow interface$ **and**
 $bot\text{-}dia :: diagram \Rightarrow interface$

where

$bot\text{-}dia (Graph\ V\ \Lambda\ E) = (\otimes v\ |\in| terminals\ (Graph\ V\ \Lambda\ E). bot\text{-}ass\ (\Lambda\ v))$
| $bot\text{-}ass (Rib\ p) = Ribbon\ p$
| $bot\text{-}ass (Exists\text{-}dia\ x\ G) = Exists\text{-}int\ x\ (bot\text{-}dia\ G)$

8.5 Proof rules for graphical diagrams

inductive

$prov\text{-}dia :: [diagram, interface, interface] \Rightarrow bool$ **and**
 $prov\text{-}com :: [command\text{-}gadget, interface, interface] \Rightarrow bool$ **and**
 $prov\text{-}ass :: assertion\text{-}gadget \Rightarrow bool$

where

$Skip: prov\text{-}ass (Rib\ p)$
| $Exists: prov\text{-}dia\ G\ -\ - \Longrightarrow prov\text{-}ass (Exists\text{-}dia\ x\ G)$
| $Basic: prov\text{-}triple\ (asn\ P,\ c,\ asn\ Q) \Longrightarrow prov\text{-}com\ (Com\ c)\ P\ Q$
| $Choice: \llbracket prov\text{-}dia\ G\ P\ Q ; prov\text{-}dia\ H\ P\ Q \rrbracket$
 $\Longrightarrow prov\text{-}com\ (Choose\text{-}dia\ G\ H)\ P\ Q$
| $Loop: prov\text{-}dia\ G\ P\ P \Longrightarrow prov\text{-}com\ (Loop\text{-}dia\ G)\ P\ P$
| $Main: \llbracket wf\text{-}dia\ G ; \bigwedge v. v \in fset\ G \hat{\wedge} V \Longrightarrow prov\text{-}ass\ (G \hat{\wedge} v);$
 $\bigwedge e. e \in set\ G \hat{\wedge} E \Longrightarrow prov\text{-}com\ (snd3\ e)$
 $(\otimes v\ |\in| fst3\ e. bot\text{-}ass\ (G \hat{\wedge} v))$
 $(\otimes v\ |\in| thd3\ e. top\text{-}ass\ (G \hat{\wedge} v)) \rrbracket$
 $\Longrightarrow prov\text{-}dia\ G\ (top\text{-}dia\ G)\ (bot\text{-}dia\ G)$

inductive-cases $main\text{-}inv: prov\text{-}dia\ (Graph\ V\ \Lambda\ E)\ P\ Q$

inductive-cases $loop\text{-}inv: prov\text{-}com\ (Loop\text{-}dia\ G)\ P\ Q$

inductive-cases $choice\text{-}inv: prov\text{-}com\ (Choose\text{-}dia\ G\ H)\ P\ Q$

inductive-cases $basic\text{-}inv: prov\text{-}com\ (Com\ c)\ P\ Q$

inductive-cases $exists\text{-}inv: prov\text{-}ass\ (Exists\text{-}dia\ x\ G)$

inductive-cases $skip\text{-}inv: prov\text{-}ass\ (Rib\ p)$

8.6 Extracting commands from diagrams

type-synonym $lin = (node + edge)\ list$

A linear extension (lin) of a diagram is a list of its nodes and edges which respects the order of those nodes and edges. That is, if an edge e goes from node v to node w , then v and e and w must have strictly increasing positions in the list.

definition $lins :: diagram \Rightarrow lin\ set$

where

$lins\ G \equiv \{\pi :: lin.$
 $(distinct\ \pi)$

$$\begin{aligned}
& \wedge (\text{set } \pi = (\text{fset } G \hat{V}) <+> (\text{set } G \hat{E})) \\
& \wedge (\forall i j v e. i < \text{length } \pi \wedge j < \text{length } \pi \wedge \pi!i = \text{Inl } v \wedge \pi!j = \text{Inr } e \\
& \quad \wedge v \in | \text{fst3 } e \longrightarrow i < j) \\
& \wedge (\forall j k w e. j < \text{length } \pi \wedge k < \text{length } \pi \wedge \pi!j = \text{Inr } e \wedge \pi!k = \text{Inl } w \\
& \quad \wedge w \in | \text{thd3 } e \longrightarrow j < k) \}
\end{aligned}$$

lemma *linsD*:

assumes $\pi \in \text{lins } G$
shows (*distinct* π)
and ($\text{set } \pi = (\text{fset } G \hat{V}) <+> (\text{set } G \hat{E})$)
and ($\forall i j v e. i < \text{length } \pi \wedge j < \text{length } \pi$
 $\wedge \pi!i = \text{Inl } v \wedge \pi!j = \text{Inr } e \wedge v \in | \text{fst3 } e \longrightarrow i < j$)
and ($\forall j k w e. j < \text{length } \pi \wedge k < \text{length } \pi$
 $\wedge \pi!j = \text{Inr } e \wedge \pi!k = \text{Inl } w \wedge w \in | \text{thd3 } e \longrightarrow j < k$)
using *assms*
apply –
apply (*unfold lins-def Collect-iff*)
apply (*elim conjE, assumption*)
done

The following lemma enables the inductive definition below to be proved monotonic. It does this by showing how one of the premises of the *coms-main* rule can be rewritten in a form that is more verbose but easier to prove monotonic.

lemma *coms-mono-helper*:

$$\begin{aligned}
& (\forall i < \text{length } \pi. \text{case-sum } (\text{coms-ass } \circ \Lambda) (\text{coms-com } \circ \text{snd3}) (\pi!i) (\text{cs!i})) \\
& = \\
& ((\forall i. i < \text{length } \pi \wedge (\exists v. (\pi!i) = \text{Inl } v) \longrightarrow \\
& \quad \text{coms-ass } (\Lambda (\text{projl } (\pi!i))) (\text{cs!i})) \wedge \\
& (\forall i. i < \text{length } \pi \wedge (\exists e. (\pi!i) = \text{Inr } e) \longrightarrow \\
& \quad \text{coms-com } (\text{snd3 } (\text{projr } (\pi!i))) (\text{cs!i}))) \\
& \textbf{apply} (\textit{intro iffI}) \\
& \textbf{apply} \textit{auto}[1] \\
& \textbf{apply} (\textit{intro allI impI, case-tac } \pi!i, \textit{auto}) \\
& \textbf{done}
\end{aligned}$$

The *coms-dia* function extracts a set of commands from a diagram. Each command in *coms-dia* G is obtained by extracting a command from each of G 's nodes and edges (using *coms-ass* or *coms-com* respectively), then picking a linear extension π of these nodes and edges (using *lins*), and composing the extracted commands in accordance with π .

inductive

coms-dia :: [*diagram, command*] \Rightarrow *bool* **and**
coms-ass :: [*assertion-gadget, command*] \Rightarrow *bool* **and**
coms-com :: [*command-gadget, command*] \Rightarrow *bool*

where

coms-skip: *coms-ass* (*Rib p*) *Skip*
| *coms-exists*: *coms-dia* G $c \Longrightarrow$ *coms-ass* (*Exists-dia* x G) c

```

| coms-basic: coms-com (Com c) c
| coms-choice:  $\llbracket \text{coms-dia } G \ c; \text{coms-dia } H \ d \rrbracket \implies$ 
   coms-com (Choose-dia G H) (Choose c d)
| coms-loop: coms-dia G c  $\implies$  coms-com (Loop-dia G) (Loop c)
| coms-main:  $\llbracket \pi \in \text{lins } (Graph \ V \ \Lambda \ E); \text{length } cs = \text{length } \pi;$ 
    $\forall i < \text{length } \pi. \text{case-sum } (coms-ass \circ \Lambda) (coms-com \circ snd\beta) (\pi!i) (cs!i) \rrbracket$ 
    $\implies \text{coms-dia } (Graph \ V \ \Lambda \ E) (\text{foldr } (;;) \ cs \ Skip)$ 

```

monos

coms-mono-helper

```

inductive-cases coms-skip-inv: coms-ass (Rib p) c
inductive-cases coms-exists-inv: coms-ass (Exists-dia x G) c
inductive-cases coms-basic-inv: coms-com (Com c') c
inductive-cases coms-choice-inv: coms-com (Choose-dia G H) c
inductive-cases coms-loop-inv: coms-com (Loop-dia G) c
inductive-cases coms-main-inv: coms-dia G c

```

end

9 Soundness for graphical diagrams

theory *Ribbons-Graphical-Soundness* **imports**

Ribbons-Graphical

More-Finite-Map

begin

We prove that the proof rules for graphical ribbon proofs are sound with respect to the rules of separation logic.

We impose an additional assumption to achieve soundness: that the Frame rule has no side-condition. This assumption is reasonable because there are several separation logics that lack such a side-condition, such as “variables-as-resource”.

We first describe how to extract proofchains from a diagram. This process is similar to the process of extracting commands from a diagram, which was described in *Ribbon-Proofs.Ribbons-Graphical*. When we extract a proofchain, we don’t just include the commands, but the assertions in between them. Our main lemma for proving soundness says that each of these proofchains corresponds to a valid separation logic proof.

9.1 Proofstate chains

When extracting a proofchain from a diagram, we need to keep track of which nodes we have processed and which ones we haven’t. A proofstate, defined below, maps a node to “Top” if it hasn’t been processed and “Bot” if it has.

datatype *topbot* = *Top* | *Bot*

type-synonym $proofstate = node \rightarrow_f topbot$

A proofstate chain contains all the nodes and edges of a graphical diagram, interspersed with proofstates that track which nodes have been processed at each point.

type-synonym $ps-chain = (proofstate, node + edge) chain$

The $next-ps$ σ function processes one node or one edge in a diagram, given the current proofstate σ . It processes a node v by replacing the mapping from v to Top with a mapping from v to Bot . It processes an edge e (whose source and target nodes are vs and ws respectively) by removing all the mappings from vs to Bot , and adding mappings from ws to Top .

fun $next-ps :: proofstate \Rightarrow node + edge \Rightarrow proofstate$

where

$next-ps \sigma (Inl v) = \sigma \ominus \{|v|\} ++_f [\{|v|\} | => Bot]$
 $| next-ps \sigma (Inr e) = \sigma \ominus fst3 e ++_f [thd3 e | => Top]$

The function $mk-ps-chain \Pi \pi$ generates from π , which is a list of nodes and edges, a proofstate chain, by interspersing the elements of π with the appropriate proofstates. The first argument Π is the part of the chain that has already been converted.

definition

$mk-ps-chain :: [ps-chain, (node + edge) list] \Rightarrow ps-chain$

where

$mk-ps-chain \equiv foldl (\lambda \Pi x. cSnoc \Pi x (next-ps (post \Pi) x))$

lemma $mk-ps-chain-preserves-length$:

fixes $\pi \Pi$

shows $chainlen (mk-ps-chain \Pi \pi) = chainlen \Pi + length \pi$

proof (*induct* π *arbitrary*: Π)

case Nil

show $?case$ **by** (*unfold* $mk-ps-chain-def$, *auto*)

next

case ($Cons x \pi$)

show $?case$

apply (*unfold* $mk-ps-chain-def$ $list.size$ *foldl.simps*)

apply (*fold* $mk-ps-chain-def$)

apply (*auto simp add*: $Cons len-snoc$)

done

qed

Distributing $mk-ps-chain$ over $(\#)$.

lemma $mk-ps-chain-cons$:

$mk-ps-chain \Pi (x \# \pi) = mk-ps-chain (cSnoc \Pi x (next-ps (post \Pi) x)) \pi$

by (*auto simp add*: $mk-ps-chain-def$)

Distributing $mk-ps-chain$ over $snoc$.

lemma *mk-ps-chain-snoc*:
mk-ps-chain Π (π @ $[x]$)
= *cSnoc* (*mk-ps-chain* Π π) x (*next-ps* (*post* (*mk-ps-chain* Π π)) x)
by (*unfold mk-ps-chain-def*, *auto*)

Distributing *mk-ps-chain* over *cCons*.

lemma *mk-ps-chain-ccons*:
fixes π Π
shows *mk-ps-chain* ($\{\!\! \{ \sigma \}\!\! \} \cdot x \cdot \Pi$) π = $\{\!\! \{ \sigma \}\!\! \} \cdot x \cdot$ *mk-ps-chain* Π π
by (*induct* π *arbitrary*: Π , *auto simp add: mk-ps-chain-cons mk-ps-chain-def*)

lemma *pre-mk-ps-chain*:
fixes Π π
shows *pre* (*mk-ps-chain* Π π) = *pre* Π
apply (*induct* π *arbitrary*: Π)
apply (*auto simp add: mk-ps-chain-def mk-ps-chain-cons pre-snoc*)
done

A chain which is obtained from the list π , has π as its list of commands. The following lemma states this in a slightly more general form, that allows for part of the chain to have already been processed.

lemma *comlist-mk-ps-chain*:
comlist (*mk-ps-chain* Π π) = *comlist* Π @ π
proof (*induct* π *arbitrary*: Π)
case *Nil*
thus *?case* **by** (*auto simp add: mk-ps-chain-def*)
next
case (*Cons* x π')
show *?case*
apply (*unfold mk-ps-chain-def foldl.simps, fold mk-ps-chain-def*)
apply (*auto simp add: Cons comlist-snoc*)
done
qed

In order to perform induction over our diagrams, we shall wish to obtain “smaller” diagrams, by removing nodes or edges. However, the syntax and well-formedness constraints for diagrams are such that although we can always remove an edge from a diagram, we cannot (in general) remove a node – the resultant diagram would not be a well-formed if an edge connected to that node.

Hence, we consider “partially-processed diagrams” (G, S) , which comprise a diagram G and a set S of nodes. S denotes the subset of G ’s initial nodes that have already been processed, and can be thought of as having been removed from G .

We now give an updated version of the *lins* G function. This was originally defined in *Ribbon-Proofs.Ribbon-Graphical*. We provide an extra parameter, S , which denotes the subset of G ’s initial nodes that shouldn’t be included

in the linear extensions.

definition $lins2 :: [node\ fset, diagram] \Rightarrow lin\ set$

where

$$\begin{aligned}
& lins2\ S\ G \equiv \{ \pi :: lin . \\
& \quad (distinct\ \pi) \\
& \quad \wedge (set\ \pi = (fset\ G^{\wedge}V - fset\ S) <+> set\ G^{\wedge}E) \\
& \quad \wedge (\forall i\ j\ v\ e. i < length\ \pi \wedge j < length\ \pi \\
& \quad \quad \wedge \pi!i = Inl\ v \wedge \pi!j = Inr\ e \wedge v \in |fst3\ e \longrightarrow i < j) \\
& \quad \wedge (\forall j\ k\ w\ e. j < length\ \pi \wedge k < length\ \pi \\
& \quad \quad \wedge \pi!j = Inr\ e \wedge \pi!k = Inl\ w \wedge w \in |thd3\ e \longrightarrow j < k) \}
\end{aligned}$$

lemma $lins2D$:

assumes $\pi \in lins2\ S\ G$

shows $distinct\ \pi$

and $set\ \pi = (fset\ G^{\wedge}V - fset\ S) <+> set\ G^{\wedge}E$
and $\bigwedge i\ j\ v\ e. \llbracket i < length\ \pi ; j < length\ \pi ;$
 $\pi!i = Inl\ v ; \pi!j = Inr\ e ; v \in |fst3\ e \rrbracket \Longrightarrow i < j$
and $\bigwedge i\ k\ w\ e. \llbracket j < length\ \pi ; k < length\ \pi ;$
 $\pi!j = Inr\ e ; \pi!k = Inl\ w ; w \in |thd3\ e \rrbracket \Longrightarrow j < k$

using $assms$

apply $(unfold\ lins2-def\ Collect-iff)$

apply $(elim\ conjE, assumption)+$

apply $blast+$

done

lemma $lins2I$:

assumes $distinct\ \pi$

and $set\ \pi = (fset\ G^{\wedge}V - fset\ S) <+> set\ G^{\wedge}E$
and $\bigwedge i\ j\ v\ e. \llbracket i < length\ \pi ; j < length\ \pi ;$
 $\pi!i = Inl\ v ; \pi!j = Inr\ e ; v \in |fst3\ e \rrbracket \Longrightarrow i < j$
and $\bigwedge j\ k\ w\ e. \llbracket j < length\ \pi ; k < length\ \pi ;$
 $\pi!j = Inr\ e ; \pi!k = Inl\ w ; w \in |thd3\ e \rrbracket \Longrightarrow j < k$

shows $\pi \in lins2\ S\ G$

using $assms$

apply $(unfold\ lins2-def\ Collect-iff, intro\ conjI)$

apply $assumption+$

apply $blast+$

done

When S is empty, the two definitions coincide.

lemma $lins-is-lins2-with-empty-S$:

$lins\ G = lins2\ \{\}\ G$

by $(unfold\ lins-def\ lins2-def, auto)$

The first proofstate for a diagram G is obtained by mapping each of its initial nodes to Top .

definition

$initial-ps :: diagram \Rightarrow proofstate$

where

$initial-ps\ G \equiv [initials\ G \mid => Top]$

The first proofstate for the partially-processed diagram G is obtained by mapping each of its initial nodes to Top , except those in S , which are mapped to Bot .

definition

$initial-ps2 :: [node\ fset, diagram] \Rightarrow proofstate$

where

$initial-ps2\ S\ G \equiv [initials\ G - S \mid => Top] ++_f [S \mid => Bot]$

When S is empty, the above two definitions coincide.

lemma *initial-ps-is-initial-ps2-with-empty-S:*

$initial-ps = initial-ps2\ \{\{\}\}$

apply (*unfold fun-eq-iff, intro allI*)

apply (*unfold initial-ps-def initial-ps2-def*)

apply *simp*

done

The following function extracts the set of proofstate chains from a diagram.

definition

$ps-chains :: diagram \Rightarrow ps-chain\ set$

where

$ps-chains\ G \equiv mk-ps-chain\ (cNil\ (initial-ps\ G))\ 'lins\ G$

The following function extracts the set of proofstate chains from a partially-processed diagram. Nodes in S are excluded from the resulting chains.

definition

$ps-chains2 :: [node\ fset, diagram] \Rightarrow ps-chain\ set$

where

$ps-chains2\ S\ G \equiv mk-ps-chain\ (cNil\ (initial-ps2\ S\ G))\ 'lins2\ S\ G$

When S is empty, the above two definitions coincide.

lemma *ps-chains-is-ps-chains2-with-empty-S:*

$ps-chains = ps-chains2\ \{\{\}\}$

apply (*unfold fun-eq-iff, intro allI*)

apply (*unfold ps-chains-def ps-chains2-def*)

apply (*fold initial-ps-is-initial-ps2-with-empty-S*)

apply (*fold lins-is-lins2-with-empty-S*)

apply *auto*

done

We now wish to describe proofstates chain that are well-formed. First, let us say that $f ++_f disjoint\ g$ is defined, when f and g have disjoint domains, as $f ++_f g$. Then, a well-formed proofstate chain consists of triples of the form $(\sigma ++_f disjoint [\{|v|\} \mid => Top], Inl\ v, \sigma ++_f disjoint [\{|v|\} \mid => Bot])$, where v is a node, or of the form $(\sigma ++_f disjoint [\{|vs|\} \mid => Bot$

], $Inr\ e, \sigma \text{ ++}_f \text{ disjoint } [\{|ws|\} \Rightarrow Top]$), where e is an edge with source and target nodes vs and ws respectively.

The definition below describes a well-formed triple; we then lift this to complete chains shortly.

definition

$wf\text{-ps-triple} :: \text{proofstate} \times (\text{node} + \text{edge}) \times \text{proofstate} \Rightarrow \text{bool}$

where

$wf\text{-ps-triple } T = (\text{case } snd3\ T \text{ of}$
 $\quad Inl\ v \Rightarrow (\exists \sigma. v \notin fmdom\ \sigma$
 $\quad \wedge\ fst3\ T = [\{|v|\} \Rightarrow Top] \text{ ++}_f \sigma$
 $\quad \wedge\ thd3\ T = [\{|v|\} \Rightarrow Bot] \text{ ++}_f \sigma)$
 $\quad | Inr\ e \Rightarrow (\exists \sigma. (fst3\ e \cup thd3\ e) \cap fmdom\ \sigma = \{\})$
 $\quad \wedge\ fst3\ T = [fst3\ e \Rightarrow Bot] \text{ ++}_f \sigma$
 $\quad \wedge\ thd3\ T = [thd3\ e \Rightarrow Top] \text{ ++}_f \sigma))$

lemma $wf\text{-ps-triple-nodeI}$:

assumes $\exists \sigma. v \notin fmdom\ \sigma \wedge$
 $\sigma1 = [\{|v|\} \Rightarrow Top] \text{ ++}_f \sigma \wedge$
 $\sigma2 = [\{|v|\} \Rightarrow Bot] \text{ ++}_f \sigma$
shows $wf\text{-ps-triple } (\sigma1, Inl\ v, \sigma2)$

using $assms\ unfolding\ wf\text{-ps-triple-def}$

by $(\text{auto simp add: } fst3\text{-simp } snd3\text{-simp } thd3\text{-simp})$

lemma $wf\text{-ps-triple-edgeI}$:

assumes $\exists \sigma. (fst3\ e \cup thd3\ e) \cap fmdom\ \sigma = \{\}$
 $\wedge\ \sigma1 = [fst3\ e \Rightarrow Bot] \text{ ++}_f \sigma$
 $\wedge\ \sigma2 = [thd3\ e \Rightarrow Top] \text{ ++}_f \sigma$
shows $wf\text{-ps-triple } (\sigma1, Inr\ e, \sigma2)$

using $assms\ unfolding\ wf\text{-ps-triple-def}$

by $(\text{auto simp add: } fst3\text{-simp } snd3\text{-simp } thd3\text{-simp})$

definition

$wf\text{-ps-chain} :: \text{ps-chain} \Rightarrow \text{bool}$

where

$wf\text{-ps-chain} \equiv \text{chain-all } wf\text{-ps-triple}$

lemma $\text{next-initial-ps2-vertex}$:

$\text{initial-ps2 } (\{|v|\} \cup S) G$
 $= \text{initial-ps2 } S G \ominus \{|v|\} \text{ ++}_f [\{|v|\} \Rightarrow Bot]$

apply $(\text{unfold } \text{initial-ps2-def})$

apply transfer

apply $(\text{auto simp add: } \text{make-map-def } \text{map-diff-def } \text{map-add-def } \text{restrict-map-def})$

done

lemma $\text{next-initial-ps2-edge}$:

assumes $G = \text{Graph } V \wedge E \text{ and } G' = \text{Graph } V' \wedge E' \text{ and}$
 $V' = V - fst3\ e \text{ and } E' = \text{removeAll } e\ E \text{ and } e \in \text{set } E \text{ and}$
 $\text{fst3 } e \subseteq S \text{ and } S \subseteq \text{initials } G \text{ and } wf\text{-dia } G$

```

shows initial-ps2 (S - fst3 e) G' =
initial-ps2 S G ⊖ fst3 e ++f [ thd3 e |=> Top ]
proof (insert assms, unfold initial-ps2-def, transfer)
fix G V Λ E G' V' E' e S
assume G-def: G = Graph V Λ E and G'-def: G' = Graph V' Λ E' and
  V'-def: V' = V - fst3 e and E'-def: E' = removeAll e E and
  e-in-E: e ∈ set E and fst-e-in-S: fst3 e |⊆| S and
  S-initials: S |⊆| initials G and wf-G: wf-dia G
have thd3 e |∩| initials G = {||}
  by (auto simp add: initials-def G-def e-in-E)
show make-map (initials G' - (S - fst3 e)) Top ++ make-map (S - fst3 e)
Bot
  = map-diff (make-map (initials G - S) Top ++ make-map S Bot) (fst3 e)
  ++ make-map (thd3 e) Top
apply (unfold make-map-def map-diff-def)
apply (unfold map-add-def restrict-map-def)
apply (unfold minus-fset)
apply (unfold fun-eq-iff initials-def)
apply (unfold G-def G'-def V'-def E'-def)
apply (unfold edges.simps vertices.simps)
apply (simp add: less-eq-fset.rep-eq e-in-E)
apply safe
apply (insert (thd3 e |∩| initials G = {||}))[1]
apply (insert S-initials, fold fset-cong)[2]
apply (unfold less-eq-fset.rep-eq initials-def filter-fset)
apply (auto simp add: G-def e-in-E)[1]
apply (auto simp add: G-def e-in-E)[1]
apply (auto simp add: G-def e-in-E)[1]
apply (insert wf-G)[1]
apply (unfold G-def vertices.simps edges.simps)
apply (drule wf-dia-inv(3))
apply (unfold acyclicity-def)
apply (metis fst-e-in-S inter-fset le-iff-inf subsetD)
apply (insert wf-G)[1]
apply (unfold G-def vertices.simps edges.simps)
apply (drule wf-dia-inv(4))
apply (drule linearityD2)
apply (fold fset-cong, unfold inter-fset fset-simps)
apply (insert e-in-E, blast)[1]
apply (insert wf-G)[1]
apply (unfold G-def vertices.simps edges.simps)
apply (drule wf-dia-inv(3))
apply (metis (lifting) e-in-E G-def empty-iff fset-simps(1))
  finter-iff linearityD(2) wf-G wf-dia-inv(4))
apply (insert wf-G)[1]
apply (unfold G-def vertices.simps edges.simps)
apply (drule wf-dia-inv(4))
apply (drule linearityD2)
apply (fold fset-cong, unfold inter-fset fset-simps)

```

```

apply (insert e-in-E, blast)[1]
apply (insert wf-G)[1]
apply (unfold G-def vertices.simps edges.simps)
apply (drule wf-dia-inv(3))
apply (metis (lifting) e-in-E G-def empty-iff fset-simps(1)
  finter-iff linearityD(2) wf-G wf-dia-inv(4))
apply (insert wf-G)
apply (unfold G-def vertices.simps edges.simps)
apply (drule wf-dia-inv(5))
apply (unfold less-eq-fset.rep-eq union-fset)
apply auto[1]
apply (drule wf-dia-inv(5))
apply (unfold less-eq-fset.rep-eq union-fset)
apply auto[1]
apply (drule wf-dia-inv(5))
apply (unfold less-eq-fset.rep-eq union-fset)
apply (auto simp add: e-in-E)[1]
apply (drule wf-dia-inv(5))
apply (unfold less-eq-fset.rep-eq union-fset)
apply (auto simp add: e-in-E)[1]
done

```

qed

lemma *next-lins2-vertex*:

```

assumes Inl v # π ∈ lins2 S G
assumes v ∉ S
shows π ∈ lins2 ({|v|} |∪| S) G

```

proof –

```

note lins2D = lins2D[OF assms(1)]

```

```

show ?thesis

```

```

proof (intro lins2I)

```

```

  show distinct π using lins2D(1) by auto

```

next

```

  have set π = set (Inl v # π) - {Inl v} using lins2D(1) by auto

```

```

  also have ... = (fset G^V - fset ({|v|} |∪| S)) <+> set G^E

```

```

    using lins2D(2) by auto

```

```

  finally show set π = (fset G^V - fset ({|v|} |∪| S)) <+> set G^E

```

```

    by auto

```

next

```

  fix i j v e

```

```

  assume i < length π j < length π π ! i = Inl v

```

```

    π ! j = Inr e v |∈| fst3 e

```

```

  thus i < j using lins2D(3)[of i+1 j+1] by auto

```

next

```

  fix j k w e

```

```

  assume j < length π k < length π π ! j = Inr e

```

```

    π ! k = Inl w w |∈| thd3 e

```

```

  thus j < k using lins2D(4)[of j+1 k+1] by auto

```

qed

qed

lemma *next-lins2-edge*:

assumes $\text{Inr } e \# \pi \in \text{lins2 } S \text{ (Graph } V \ \Lambda \ E)$

and $vs \sqsubseteq S$

and $e = (vs, c, ws)$

shows $\pi \in \text{lins2 } (S - vs) \text{ (Graph } (V - vs) \ \Lambda \ (\text{removeAll } e \ E))$

proof –

note $\text{lins2D} = \text{lins2D}[\text{OF } \text{assms}(1)]$

show *?thesis*

proof (*intro lins2I, unfold vertices.simps edges.simps*)

show *distinct* π

using $\text{lins2D}(1)$ **by** *auto*

next

show $\text{set } \pi = (\text{fset } (V - vs) - \text{fset } (S - vs))$

$\langle + \rangle \text{ set } (\text{removeAll } e \ E)$

apply (*insert lins2D(1) lins2D(2) assms(2)*)

apply (*unfold assms(3) vertices.simps edges.simps less-eq-fset.rep-eq, simp*)

apply (*unfold diff-diff-eq*)

proof –

have $\forall a \ aa \ b.$

$\text{insert } (\text{Inr } (vs, c, ws)) (\text{set } \pi) = (\text{fset } V - \text{fset } S) \langle + \rangle \text{ set } E \longrightarrow$

$\text{fset } vs \subseteq \text{fset } S \longrightarrow$

$\text{Inr } (vs, c, ws) \notin \text{set } \pi \longrightarrow$

$\text{distinct } \pi \longrightarrow (a, aa, b) \in \text{set } E \longrightarrow \text{Inr } (a, aa, b) \notin \text{set } \pi \longrightarrow b = ws$

by (*metis (lifting) InrI List.set-simps(2)*)

prod.inject set-ConsD sum.simps(2))

moreover have $\forall a \ aa \ b.$

$\text{insert } (\text{Inr } (vs, c, ws)) (\text{set } \pi) = (\text{fset } V - \text{fset } S) \langle + \rangle \text{ set } E \longrightarrow$

$\text{fset } vs \subseteq \text{fset } S \longrightarrow$

$\text{Inr } (vs, c, ws) \notin \text{set } \pi \longrightarrow$

$\text{distinct } \pi \longrightarrow (a, aa, b) \in \text{set } E \longrightarrow \text{Inr } (a, aa, b) \notin \text{set } \pi \longrightarrow aa = c$

by (*metis (lifting) InrI List.set-simps(2)*)

prod.inject set-ConsD sum.simps(2))

moreover have $\forall x. \text{insert } (\text{Inr } (vs, c, ws)) (\text{set } \pi) = (\text{fset } V - \text{fset } S) \langle + \rangle \text{ set } E \longrightarrow$

$\text{fset } vs \subseteq \text{fset } S \longrightarrow$

$\text{Inr } (vs, c, ws) \notin \text{set } \pi \longrightarrow$

$\text{distinct } \pi \longrightarrow x \in \text{set } \pi \longrightarrow x \in (\text{fset } V - \text{fset } S) \langle + \rangle \text{ set } E - \{(vs, c, ws)\}$

apply (*unfold insert-is-Un[of - set π]*)

apply (*fold assms(3)*)

apply *clarify*

apply (*subgoal-tac set $\pi = ((\text{fset } V - \text{fset } S) \langle + \rangle \text{ set } E) - \{\text{Inr } e\}$*)

by *auto*

ultimately show $\text{Inr } (vs, c, ws) \notin \text{set } \pi \wedge \text{distinct } \pi \implies$

$\text{insert } (\text{Inr } (vs, c, ws)) (\text{set } \pi) = (\text{fset } V - \text{fset } S) \langle + \rangle \text{ set } E \implies$

```

    fset vs  $\subseteq$  fset S  $\implies$  set  $\pi = (fset V - fset S) <+>$  set E - {(vs, c, ws)}
  by blast
  qed
next
  fix i j v e
  assume i < length  $\pi$  j < length  $\pi$   $\pi ! i = Inl v$ 
     $\pi ! j = Inr e v \mid \in \mid fst3 e$ 
  thus i < j using lins2D(3)[of i+1 j+1] by auto
next
  fix j k w e
  assume j < length  $\pi$  k < length  $\pi$   $\pi ! j = Inr e$ 
     $\pi ! k = Inl w w \mid \in \mid thd3 e$ 
  thus j < k using lins2D(4)[of j+1 k+1] by auto
  qed
qed

```

We wish to prove that every proofstate chain that can be obtained from a linear extension of G is well-formed and has as its final proofstate that state in which every terminal node in G is mapped to Bot .

We first prove this for partially-processed diagrams, for then the result for ordinary diagrams follows as an easy corollary.

We use induction on the size of the partially-processed diagram. The size of a partially-processed diagram (G, S) is defined as the number of nodes in G , plus the number of edges, minus the number of nodes in S .

lemma *wf-chains2*:

```

  fixes k
  assumes S  $\mid \subseteq \mid$  initials G
    and wf-dia G
    and  $\Pi \in ps-chains2 S G$ 
    and fcard  $G \hat{=} V + length G \hat{=} E = k + fcard S$ 
  shows wf-ps-chain  $\Pi \wedge (post \Pi = [ terminals G \mid => Bot ])$ 
using assms
proof (induct k arbitrary: S G  $\Pi$ )
  case 0
  obtain V  $\Lambda E$  where G-def: G = Graph V  $\Lambda E$  by (metis diagram.exhaust)
  have S  $\mid \subseteq \mid$  V
    using 0.prem1 initials-in-vertices[of G]
    by (auto simp add: G-def)
  have fcard V  $\leq$  fcard S
    using 0.prem4
    by (unfold G-def, auto)
  from fcard-seteq[OF  $\langle S \mid \subseteq \mid V \rangle$  this] have S = V by auto
  hence E = [] using 0.prem4 by (unfold G-def, auto)
  have initials G = V
    by (unfold G-def  $\langle E = [] \rangle$ , rule no-edges-imp-all-nodes-initial)
  have terminals G = V
    by (unfold G-def  $\langle E = [] \rangle$ , rule no-edges-imp-all-nodes-terminal)
  have {} <+> {} = {} by auto

```

```

have lins2  $S$   $G = \{ [] \}$ 
apply (unfold  $G$ -def  $\langle S=V \rangle \langle E=[] \rangle$ )
apply (unfold lins2-def, auto simp add:  $\langle \{ \} \langle + \rangle \{ \} = \{ \} \rangle$ )
done
hence  $\Pi$ -def:  $\Pi = \{ \text{initial-ps2 } S \ G \}$ 
  using  $0$ .prems(3)
  by (auto simp add: ps-chains2-def mk-ps-chain-def)
show ?case
apply (intro conjI)
apply (unfold  $\Pi$ -def wf-ps-chain-def, auto)
apply (unfold post.simps initial-ps2-def  $\langle \text{initials } G = V \rangle \langle \text{terminals } G = V \rangle$ )
apply (unfold  $\langle S=V \rangle$ )
apply (subgoal-tac  $V - V = \{ [] \}$ , simp-all)
done
next
case (Suc  $k$ )
obtain  $V \ \Lambda \ E$  where  $G$ -def:  $G = \text{Graph } V \ \Lambda \ E$  by (metis diagram.exhaust)
from Suc.prems(3) obtain  $\pi$  where
   $\Pi$ -def:  $\Pi = \text{mk-ps-chain } \{ \text{initial-ps2 } S \ G \}$  and
   $\pi$ -in:  $\pi \in \text{lins2 } S \ G$ 
  by (auto simp add: ps-chains2-def)
note lins2 = lins2D[OF  $\pi$ -in]
have  $S \subseteq V$ 
  using Suc.prems(1) initials-in-vertices[of  $G$ ]
  by (auto simp add:  $G$ -def)
show ?case
proof (cases  $\pi$ )
  case Nil
  from  $\pi$ -in have  $V = S \ E = []$ 
  apply ( $-$ , unfold  $\langle \pi = [] \rangle$  lins2-def, simp-all)
  apply (unfold empty-eq-Plus-conv)
  apply (unfold  $G$ -def vertices.simps edges.simps, auto)
  by (metis  $\langle S \subseteq V \rangle$  less-eq-fset.rep-eq subset-antisym)

  with Suc.prems(4) have False by (simp add:  $G$ -def)
  thus ?thesis by auto
next
case (Cons  $x \ \pi'$ )
note  $\pi$ -def = this
show ?thesis
proof (cases  $x$ )
  case (Inl  $v$ )
  note  $x$ -def = this

  have  $v \notin S \ \wedge \ v \in V$ 
  apply (subgoal-tac  $v \in \text{fset } V - \text{fset } S$ )
  apply (simp)
  apply (subgoal-tac Inl  $v \in (\text{fset } V - \text{fset } S) \langle + \rangle \text{set } E$ )
  apply (metis Inl-inject Inr-not-Inl PlusE)

```

```

apply (metis lins2(1) lins2(2) Cons G-def Inl distinct.simps(2)
  distinct-length-2-or-more edges.simps vertices.simps)
done
hence v-notin-S: v ∉ S and v-in-V: v ∈ V by auto

have v-initial-not-S: v ∈ initials G - S
apply (simp only: G-def initials-def vertices.simps edges.simps)
apply (simp only: fminus-iff)
apply (simp only: conj-commute, intro conjI, rule v-notin-S)
apply (subgoal-tac
  v ∈ fset (ffilter (λv. ∀ e ∈ set E. v ∉ thd3 e) V))
apply simp
apply (simp only: filter-fset, simp, simp only: conj-commute)
apply (intro conjI ballI notI)
apply (insert v-in-V, simp)
proof -
  fix e :: edge
  assume v ∈ fset (thd3 e)
  then have v ∈ (thd3 e) by auto
  assume e ∈ set E
  hence Inr e ∈ set π using lins2(2) by (auto simp add: G-def)
  then obtain j where
    j < length π 0 < length π π!j = Inr e π!0 = Inl v
  by (metis π-def x-def in-set-conv-nth length-pos-if-in-set nth-Cons-0)
  with lins2(4)[OF this ⟨v ∈ (thd3 e)⟩] show False by auto
qed

define S' where S' = {|v|} |∪| S

define Π' where Π' = mk-ps-chain { initial-ps2 S' G } π'
hence pre-Π': pre Π' = initial-ps2 S' G
by (metis pre.simps(1) pre-mk-ps-chain)

define σ where σ = [ initials G - ({|v|} |∪| S) |=> Top ] ++_f [ S |=> Bot
]

have wf-ps-chain Π' ∧ (post Π' = [terminals G |=> Bot])
proof (intro Suc.hyps[of S'])
  show S' |⊆| initials G
  apply (unfold S'-def, auto)
  apply (metis fminus-iff v-initial-not-S)
  by (metis Suc.prem(1) fset-rev-mp)
next
  show wf-dia G by (rule Suc.prem(2))
next
  show Π' ∈ ps-chains2 S' G
  apply (unfold ps-chains2-def Π'-def)
  apply (intro imageI)
  apply (unfold S'-def)

```

```

apply (intro next-lins2-vertex)
apply (fold x-def, fold  $\pi$ -def)
apply (rule  $\pi$ -in)
by (metis v-notin-S)
next
show fcard  $G^{\wedge}V + \text{length } G^{\wedge}E = k + \text{fcard } S'$ 
apply (unfold S'-def)
by (auto simp add: Suc.premis(4) fcard-finsert-disjoint[OF v-notin-S])
qed
hence
  wf- $\Pi'$ : wf-ps-chain  $\Pi'$  and
  post- $\Pi'$ : post  $\Pi' = [\text{terminals } G \mid \Rightarrow \text{Bot}]$ 
by auto

show ?thesis
proof (intro conjI)
  have 1: fmdom [  $\{|v|\} \mid \Rightarrow \text{Bot}$  ]
    | $\cap$ | fmdom ( [ initials  $G - (\{|v|\} \cup S) \mid \Rightarrow \text{Top}$  ] ++f
  [  $S \mid \Rightarrow \text{Bot}$  ] ) = {||}
  by (metis (no-types) fdom-make-fmap fmdom-add
    bot-least funion-iff finter-finsert-left le-iff-inf
    fminus-iff finsert-fsubset sup-ge1 v-initial-not-S)
  show wf-ps-chain  $\Pi$ 
  using [[unfold-abs-def = false]]
  apply (simp only:  $\Pi$ -def  $\pi$ -def x-def mk-ps-chain-cons)
  apply simp
  apply (unfold mk-ps-chain-ccons)
  apply (fold next-initial-ps2-vertex S'-def)
  apply (fold  $\Pi'$ -def)
  apply (unfold wf-ps-chain-def chain-all.simps conj-commute)
  apply (intro conjI)
  apply (fold wf-ps-chain-def, rule wf- $\Pi'$ )
  apply (intro wf-ps-triple-nodeI exI[of -  $\sigma$ ] conjI)
  apply (unfold  $\sigma$ -def fmdom-add fdom-make-fmap)
  apply (metis finsertI1 fminus-iff funion-iff v-notin-S)
  apply (unfold pre- $\Pi'$  initial-ps2-def S'-def)
  apply (unfold fmap-add-commute[OF 1])
  apply (unfold fmadd-assoc)
  apply (fold fmadd-assoc[of - [  $S \mid \Rightarrow \text{Bot}$  ]])
  apply (unfold make-fmap-union sup.commute[of  $\{|v|\}$ ])
  apply (unfold fminus-funion)
  using v-initial-not-S apply auto
  by (metis (opaque-lifting, no-types) finsert-absorb finsert-fminus-single fin-
  ter-fminus
    inf-commute inf-idem v-initial-not-S)
next
show post  $\Pi = [\text{terminals } G \mid \Rightarrow \text{Bot}]$ 
apply (unfold  $\Pi$ -def  $\pi$ -def x-def mk-ps-chain-cons, simp)
apply (unfold mk-ps-chain-ccons post.simps)

```

```

apply (fold next-initial-ps2-vertex S'-def)
apply (fold  $\Pi'$ -def, rule post- $\Pi'$ )
done
qed
next
case (Inr e)
note x-def = this
define vs where vs = fst3 e
define ws where ws = thd3 e

obtain c where e-def: e = (vs, c, ws)
by (metis vs-def ws-def fst3-simp thd3-simp prod-cases3)

have linearity E and acyclicity E and
  e-in-V:  $\bigwedge e. e \in \text{set } E \implies \text{fst3 } e \mid \cup \mid \text{thd3 } e \mid \subseteq V$ 
by (insert Suc.prem(2) wf-dia-inv, unfold G-def, blast)+
note lin = linearityD[OF this(1)]

have acy:  $\bigwedge e. e \in \text{set } E \implies \text{fst3 } e \mid \cap \mid \text{thd3 } e = \{\mid\}$ 
apply (fold fset-cong, insert  $\langle$ acyclicity E $\rangle$ )
apply (unfold acyclicity-def acyclic-def, auto)
done

note lins = lins2D[OF  $\pi$ -in]

have e-in-E: e  $\in$  set E
apply (subgoal-tac set  $\pi = (\text{fset } G^{\wedge}V - \text{fset } S) \langle + \rangle \text{set } G^{\wedge}E$ )
apply (unfold  $\pi$ -def x-def G-def edges.simps, auto)[1]
apply (simp add: lins(2))
done

have vs-in-S: vs  $\mid \subseteq \mid S$ 
apply (insert e-in-V[OF e-in-E])
apply (unfold less-eq-fset.rep-eq)
apply (intro subsetI)
apply (unfold vs-def)
apply (rule ccontr)
apply (subgoal-tac x  $\in$  fset V)
prefer 2
apply (auto)
proof -
  fix v
  assume a: v  $\in$  fset (fst3 e)
  assume v  $\notin$  fset S and v  $\in$  fset V
  hence Inl v  $\in$  set  $\pi$ 
  by (metis (lifting) DiffI G-def InlI lins(2) vertices.simps)
  then obtain i where
    i < length  $\pi$  0 < length  $\pi$   $\pi!$ i = Inl v  $\pi!$ 0 = Inr e
  by (metis Cons Inr in-set-conv-nth length-pos-if-in-set nth-Cons-0)

```

```

from lins(3)[OF this] show False by (auto simp add: a)
qed

have ws | $\cap$ | (initials G) = {||}
apply (insert e-in-V[OF e-in-E])
apply (unfold initials-def less-eq-fset.rep-eq, fold fset-cong)
apply (unfold ws-def G-def, auto simp add: e-in-E)
done

define S' where S' = S - vs
define V' where V' = V - vs
define E' where E' = removeAll e E
define G' where G' = Graph V'  $\wedge$  E'

define  $\Pi'$  where  $\Pi'$  = mk-ps-chain { initial-ps2 S' G' }  $\pi'$ 
hence pre- $\Pi'$ : pre  $\Pi'$  = initial-ps2 S' G'
by (metis pre.simps(1) pre-mk-ps-chain)

define  $\sigma$  where  $\sigma$  = [ initials G - S | $\Rightarrow$  Top ] ++f [ S - vs | $\Rightarrow$  Bot ]

have next-initial-ps2: initial-ps2 S' G'
  = initial-ps2 S G  $\ominus$  vs ++f [ws | $\Rightarrow$  Top]
using next-initial-ps2-edge[OF G-def - - e-in-E - Suc.prem(1)
  Suc.prem(2)] G'-def E'-def vs-def ws-def V'-def vs-in-S S'-def
by auto

have wf-ps-chain  $\Pi' \wedge$  post  $\Pi'$  = [ terminals G' | $\Rightarrow$  Bot ]
proof (intro Suc.hyps[of S'])
  show S' | $\subseteq$ | initials G'
  apply (insert Suc.prem(1))
  apply (unfold G'-def G-def initials-def)
  apply (unfold less-eq-fset.rep-eq S'-def E'-def V'-def)
  apply auto
done
next
from Suc.prem(2) have wf-dia (Graph V'  $\wedge$  E')
  by (unfold G-def)
note wf-G = wf-dia-inv[OF this]
show wf-dia G'
apply (unfold G'-def V'-def E'-def)
apply (insert wf-G e-in-E vs-in-S Suc.prem(1))
apply (unfold vs-def)
apply (intro wf-dia)
apply (unfold linearity-def initials-def G-def)
apply (fold fset-cong, unfold less-eq-fset.rep-eq)
apply (simp, simp)
apply (unfold acyclicity-def, rule acyclic-subset)
apply (auto simp add: distinct-removeAll)
apply (metis (lifting) IntI empty-iff)

```

```

done
next
show  $\Pi' \in ps-chains2\ S'\ G'$ 
apply (unfold  $\Pi$ -def  $\Pi'$ -def  $ps-chains2$ -def)
apply (intro imageI)
apply (unfold  $S'$ -def  $G'$ -def  $V'$ -def  $E'$ -def)
apply (intro next-lins2-edge)
apply (metis  $\pi$ -def  $G$ -def  $x$ -def  $\pi$ -in)
by (simp only:  $vs$ -in- $S$   $e$ -def)+
next
have  $vs \subseteq V$  by (metis (lifting)  $\langle S \subseteq V \rangle$  order-trans  $vs$ -in- $S$ )
have distinct  $E$  using  $\langle linearity\ E \rangle$  linearity-def by auto
show  $fcard\ G' \wedge V + length\ G' \wedge E = k + fcard\ S'$ 
apply (insert Suc.prem5(4))
apply (unfold  $G$ -def  $G'$ -def vertices.simps edges.simps)
apply (unfold  $V'$ -def  $E'$ -def  $S'$ -def)
apply (unfold fcard-union-fsubset[OF  $\langle vs \subseteq V \rangle$ ])
apply (unfold fcard-union-fsubset[OF  $\langle vs \subseteq S \rangle$ ])
apply (fold distinct-remove1-removeAll[OF  $\langle distinct\ E \rangle$ ])
apply (unfold length-remove1)
apply (simp add:  $e$ -in- $E$ )
apply (drule arg-cong[of - -  $\lambda x. x - fcard\ vs - 1$ ])
apply (subst (asm) add-diff-assoc2[symmetric])
apply (simp add: fcard-mono[OF  $\langle vs \subseteq V \rangle$ ])
apply (subst add-diff-assoc, insert length-pos-if-in-set[OF  $e$ -in- $E$ ], arith,
auto)
apply (subst add-diff-assoc, auto simp add: fcard-mono[OF  $\langle vs \subseteq S \rangle$ ])
done
qed
hence
wf- $\Pi'$ : wf-ps-chain  $\Pi'$  and
post- $\Pi'$ : post  $\Pi' = [ terminals\ G' \mid \Rightarrow Bot ]$ 
by auto

have terms-same: terminals  $G = terminals\ G'$ 
apply (unfold  $G'$ -def  $G$ -def terminals-def edges.simps vertices.simps)
apply (unfold  $E'$ -def  $V'$ -def)
apply (fold fset-cong, auto simp add:  $e$ -in- $E$   $vs$ -def)
done

have 1: fmdom [fst3  $e \mid \Rightarrow Bot$ ]  $\cap$ 
fmdom([ffilter ( $\lambda v. \forall e \in set\ E. v \notin thd3\ e$ )  $V - S \mid \Rightarrow Top$ ]
++f [  $S - fst3\ e \mid \Rightarrow Bot$  ]) = {}
apply (unfold fmdom-add fdom-make-fmap)
apply (fold fset-cong)
apply auto
apply (metis in-mono less-eq-fset.rep-eq  $vs$ -def  $vs$ -in- $S$ )
done

```

```

show ?thesis
proof (intro conjI)
  show wf-ps-chain  $\Pi$ 
  using [[unfold-abs-def = false]]
  apply (unfold  $\Pi$ -def  $\pi$ -def  $x$ -def mk-ps-chain-cons)
  apply simp
  apply (unfold mk-ps-chain-ccons)
  apply (fold vs-def ws-def)
  apply (fold next-initial-ps2)
  apply (fold  $\Pi'$ -def)
  apply (unfold wf-ps-chain-def chain-all.simps conj-commute)
  apply (intro conjI)
  apply (fold wf-ps-chain-def)
  apply (rule wf- $\Pi^\wedge$ )
  apply (intro wf-ps-triple-edgeI exI[of -  $\sigma$ ])
  apply (unfold e-def fst3-simp thd3-simp  $\sigma$ -def, intro conjI)
  apply (insert Suc.prem1)
  apply (unfold pre- $\Pi'$  initial-ps2-def initials-def)
  apply (insert vs-in-S acy[OF e-in-E])
  apply (fold fset-cong)
  apply (unfold less-eq-fset.rep-eq)[1]
  apply (unfold G-def G'-def vs-def ws-def V'-def E'-def S'-def)
  apply (unfold vertices.simps edges.simps)
  apply (unfold fmap-add-commute[OF 1])
  apply (fold fmadd-assoc)
  apply (unfold make-fmap-union)
  apply (auto simp add: fdom-make-fmap e-in-E)[1]
  apply simp
  apply (unfold fmadd-assoc)
  apply (unfold make-fmap-union)
  apply (metis (lifting) funion-absorb2 vs-def vs-in-S)
  apply (intro arg-cong2[of - - [ S - fst3 e  $\Rightarrow$  Bot ]
    [ S - fst3 e  $\Rightarrow$  Bot ] (++)])
  apply (intro arg-cong2[of - - Top Top make-fmap])
  defer 1
  apply (simp, simp)
  apply (fold fset-cong)
  apply (unfold less-eq-fset.rep-eq, simp)
  apply (elim conjE)
  apply (intro set-eqI iffI, simp-all)
  apply (elim conjE, intro disjI conjI ballI, simp)
  apply (case-tac ea=e, simp-all)
  apply (elim disjE conjE, intro conjI ballI impI, simp-all)
  apply (insert e-in-E lin(2))[1]
  apply (subst (asm) (2) fset-cong[symmetric])
  apply (elim conjE)
  apply (subst (asm) inter-fset)
  apply (subst (asm) fset-simps)
  apply (insert disjoint-iff-not-equal)[1]

```

```

apply blast
apply (metis G-def Suc(3) e-in-E subsetD less-eq-fset.rep-eq wf-dia-inv')
prefer 2
apply (metis (lifting) IntI Suc(2) ‹ws |∩| initials G = {||}›
  empty-iff fset-simps(1) in-mono inter-fset less-eq-fset.rep-eq ws-def)
apply auto
done
next
show post Π = [terminals G |=> Bot]
apply (unfold Π-def π-def x-def mk-ps-chain-cons)
apply simp
apply (unfold mk-ps-chain-ccons post.simps)
apply (fold vs-def ws-def)
apply (fold next-initial-ps2)
apply (fold Π'-def)
apply (unfold terms-same)
apply (rule post-Π')
done
qed
qed
qed
qed

```

corollary *wf-chains*:

```

assumes wf-dia G
assumes  $\Pi \in ps-chains\ G$ 
shows wf-ps-chain Π ∧ post Π = [ terminals G |=> Bot ]
apply (intro wf-chains2[of {||}], insert assms(2))
by (auto simp add: assms(1) ps-chains-is-ps-chains2-with-empty-S fcard-fempty)

```

9.2 Interface chains

type-synonym *int-chain* = (*interface, assertion-gadget + command-gadget*) *chain*

An interface chain is similar to a proofstate chain. However, where a proofstate chain talks about nodes and edges, an interface chain talks about the assertion-gadgets and command-gadgets that label those nodes and edges in a diagram. And where a proofstate chain talks about proofstates, an interface chain talks about the interfaces obtained from those proofstates. The following functions convert a proofstate chain into an interface chain.

definition

```

 $ps-to-int :: [diagram, proofstate] \Rightarrow interface$ 
where
 $ps-to-int\ G\ \sigma \equiv$ 
 $\bigotimes v \in |fndom\ \sigma.\ case-topbot\ top-ass\ bot-ass\ (lookup\ \sigma\ v)\ (G \wedge \Lambda\ v)$ 

```

definition

```

 $ps-chain-to-int-chain :: [diagram, ps-chain] \Rightarrow int-chain$ 
where

```

ps-chain-to-int-chain $G \Pi \equiv$
 $\text{chainmap } (\text{ps-to-int } G) ((\text{case-sum } (\text{Inl} \circ G \wedge \Lambda) (\text{Inr} \circ \text{snd}\beta))) \Pi$

lemma *ps-chain-to-int-chain-simp*:

ps-chain-to-int-chain $(\text{Graph } V \Lambda E) \Pi =$
 $\text{chainmap } (\text{ps-to-int } (\text{Graph } V \Lambda E)) ((\text{case-sum } (\text{Inl} \circ \Lambda) (\text{Inr} \circ \text{snd}\beta))) \Pi$
by (*simp add: ps-chain-to-int-chain-def*)

9.3 Soundness proof

We assume that *wr-com* always returns $\{\}$. This is equivalent to changing our axiomatization of separation logic such that the frame rule has no side-condition. One way to obtain a separation logic lacking a side-condition on its frame rule is to use variables-as- resource.

We proceed by induction on the proof rules for graphical diagrams. We show that: (1) if a diagram G is provable w.r.t. interfaces P and Q , then P and Q are the top and bottom interfaces of G , and that the Hoare triple $(\text{asn } P, c, \text{asn } Q)$ is provable for each command c that can be extracted from G ; (2) if a command-gadget C is provable w.r.t. interfaces P and Q , then the Hoare triple $(\text{asn } P, c, \text{asn } Q)$ is provable for each command c that can be extracted from C ; and (3) if an assertion-gadget A is provable, and if the top and bottom interfaces of A are P and Q respectively, then the Hoare triple $(\text{asn } P, c, \text{asn } Q)$ is provable for each command c that can be extracted from A .

lemma *soundness-graphical-helper*:

assumes *no-var-interference*: $\bigwedge c. \text{wr-com } c = \{\}$

shows

$(\text{prov-dia } G P Q \longrightarrow$
 $(P = \text{top-dia } G \wedge Q = \text{bot-dia } G \wedge$
 $(\forall c. \text{coms-dia } G c \longrightarrow \text{prov-triple } (\text{asn } P, c, \text{asn } Q))))$
 $\wedge (\text{prov-com } C P Q \longrightarrow$
 $(\forall c. \text{coms-com } C c \longrightarrow \text{prov-triple } (\text{asn } P, c, \text{asn } Q)))$
 $\wedge (\text{prov-ass } A \longrightarrow$
 $(\forall c. \text{coms-ass } A c \longrightarrow \text{prov-triple } (\text{asn } (\text{top-ass } A), c, \text{asn } (\text{bot-ass } A))))$

proof (*induct rule: prov-dia-prov-com-prov-ass.induct*)

case (*Skip* p)

thus *?case*

apply (*intro allI impI, elim conjE coms-skip-inv*)

apply (*auto simp add: prov-triple.skip*)

done

next

case (*Exists* $G P Q x$)

thus *?case*

apply (*intro allI impI, elim conjE coms-exists-inv*)

apply (*auto simp add: prov-triple.exists*)

done

next

```

case (Basic P c Q)
thus ?case
by (intro allI impI, elim conjE coms-basic-inv, auto)
next
case (Choice G P Q H)
thus ?case
apply (intro allI impI, elim conjE coms-choice-inv)
apply (auto simp add: prov-triple.choose)
done
next
case (Loop G P)
thus ?case
apply (intro allI impI, elim conjE coms-loop-inv)
apply (auto simp add: prov-triple.loop)
done
next
case (Main G)
thus ?case
apply (intro conjI)
apply (simp, simp)
apply (intro allI impI)
apply (elim coms-main-inv, simp)
proof –
  fix c V  $\Lambda$  E
  fix  $\pi::lin$ 
  fix cs::command list
  assume wf-G: wf-dia (Graph V  $\Lambda$  E)
  assume  $\bigwedge v. v \in fset\ V \implies \forall c. coms-ass\ (\Lambda\ v)\ c \longrightarrow$ 
    prov-triple (asn (top-ass ( $\Lambda\ v$ )), c, asn (bot-ass ( $\Lambda\ v$ )))
  hence prov-vertex:  $\bigwedge v\ c\ P\ Q\ F. \llbracket coms-ass\ (\Lambda\ v)\ c; v \in fset\ V;$ 
    P = (top-ass ( $\Lambda\ v$ )  $\otimes$  F) ; Q = (bot-ass ( $\Lambda\ v$ )  $\otimes$  F)  $\rrbracket$ 
     $\implies prov-triple\ (asn\ P,\ c,\ asn\ Q)$ 
  by (auto simp add: prov-triple.frame no-var-interference)
  assume  $\bigwedge e. e \in set\ E \implies \forall c. coms-com\ (snd3\ e)\ c \longrightarrow prov-triple$ 
    (asn ( $\bigotimes v|e|fst3\ e. bot-ass\ (\Lambda\ v)$ ), c, asn ( $\bigotimes v|e|thd3\ e. top-ass\ (\Lambda\ v)$ ))
  hence prov-edge:  $\bigwedge e\ c\ P\ Q\ F. \llbracket e \in set\ E ; coms-com\ (snd3\ e)\ c ;$ 
    P = (( $\bigotimes v|e|fst3\ e. bot-ass\ (\Lambda\ v)$ )  $\otimes$  F) ;
    Q = (( $\bigotimes v|e|thd3\ e. top-ass\ (\Lambda\ v)$ )  $\otimes$  F)  $\rrbracket$ 
     $\implies prov-triple\ (asn\ P,\ c,\ asn\ Q)$ 
  by (auto simp add: prov-triple.frame no-var-interference)
  assume len-cs: length cs = length  $\pi$ 
  assume  $\forall i < length\ \pi.$ 
    case-sum (coms-ass  $\circ$   $\Lambda$ ) (coms-com  $\circ$  snd3) ( $\pi\ !\ i$ ) (cs ! i)
  hence  $\pi-cs: \bigwedge i. i < length\ \pi \implies$ 
    case-sum (coms-ass  $\circ$   $\Lambda$ ) (coms-com  $\circ$  snd3) ( $\pi\ !\ i$ ) (cs ! i) by auto
  assume G-def: G = Graph V  $\Lambda$  E
  assume c-def: c = foldr (:) cs Skip
  assume  $\pi-lin: \pi \in lins\ (Graph\ V\ \Lambda\ E)$ 

```

```

note  $lins = linsD[OF \pi\text{-}lin]$ 

define  $\Pi$  where  $\Pi = mk\text{-}ps\text{-}chain \{ initial\text{-}ps \ G \} \ \pi$ 

have  $\Pi \in ps\text{-}chains \ G$  by (simp add:  $\pi\text{-}lin \ \Pi\text{-}def \ ps\text{-}chains\text{-}def \ G\text{-}def$ )
hence  $1: post \ \Pi = [ terminals \ G \ \Rightarrow \ Bot ]$ 
and  $2: chain\text{-}all \ wf\text{-}ps\text{-}triple \ \Pi$ 
by (insert wf\text{-}chains \ G\text{-}def \ wf\text{-}G, auto simp add: wf\text{-}ps\text{-}chain\text{-}def)

show prov\text{-}triple (asn ( $\otimes v \in |initials (Graph \ V \ \Lambda \ E)$ ). top\text{-}ass ( $\Lambda \ v$ )),
foldr ( $;;$ ) cs Skip, asn ( $\otimes v \in |terminals (Graph \ V \ \Lambda \ E)$ ). bot\text{-}ass ( $\Lambda \ v$ )))
using [unfold\text{-}abs\text{-}def = false]
apply (intro seq\text{-}fold[of - ps\text{-}chain\text{-}to\text{-}int\text{-}chain \ G \ \Pi])
apply (unfold len\text{-}cs)
apply (unfold ps\text{-}chain\text{-}to\text{-}int\text{-}chain\text{-}def chainmap\text{-}preserves\text{-}length \ \Pi\text{-}def)
apply (unfold mk\text{-}ps\text{-}chain\text{-}preserves\text{-}length, simp)
apply (unfold pre\text{-}chainmap post\text{-}chainmap)
apply (unfold pre\text{-}mk\text{-}ps\text{-}chain pre.simps)
apply (fold \ \Pi\text{-}def, unfold 1)
apply (unfold initial\text{-}ps\text{-}def)
apply (unfold ps\text{-}to\text{-}int\text{-}def)
apply (unfold fdom\text{-}make\text{-}fmap)
apply (unfold G\text{-}def labelling.simps, fold G\text{-}def)
apply (subgoal\text{-}tac  $\forall v \in fset (initials \ G)$ . top\text{-}ass ( $\Lambda \ v$ ) =
case\text{-}topbot top\text{-}ass bot\text{-}ass (lookup [ initials \ G \ \Rightarrow \ Top ] \ v) (\Lambda \ v)))
apply (unfold iter\text{-}hcomp\text{-}cong, simp)
apply (metis lookup\text{-}make\text{-}fmap topbot.simps(3))
apply (subgoal\text{-}tac  $\forall v \in fset (terminals \ G)$ . bot\text{-}ass ( $\Lambda \ v$ ) =
case\text{-}topbot top\text{-}ass bot\text{-}ass (lookup [ terminals \ G \ \Rightarrow \ Bot ] \ v) (\Lambda \ v)))
apply (unfold iter\text{-}hcomp\text{-}cong, simp)
apply (metis lookup\text{-}make\text{-}fmap topbot.simps(4), simp)
apply (unfold G\text{-}def, fold ps\text{-}chain\text{-}to\text{-}int\text{-}chain\text{-}simp \ G\text{-}def)
proof -
fix  $i$ 
assume  $i < length \ \pi$ 
hence  $i < chainlen \ \Pi$ 
by (metis \ \Pi\text{-}def add\text{-}0\text{-}left chainlen.simps(1)
mk\text{-}ps\text{-}chain\text{-}preserves\text{-}length)
hence  $wf\text{-}\Pi i: wf\text{-}ps\text{-}triple (nthtriple \ \Pi \ i)$ 
by (insert 2, simp add: chain\text{-}all\text{-}nthtriple)
show prov\text{-}triple (asn (fst3 (nthtriple (ps\text{-}chain\text{-}to\text{-}int\text{-}chain \ G \ \Pi) \ i)),
cs ! i, asn (thd3 (nthtriple (ps\text{-}chain\text{-}to\text{-}int\text{-}chain \ G \ \Pi) \ i))))
apply (unfold ps\text{-}chain\text{-}to\text{-}int\text{-}chain\text{-}def)
apply (unfold nthtriple\text{-}chainmap[OF  $i < chainlen \ \Pi$ ])
apply (unfold fst3\text{-}simp thd3\text{-}simp)
proof (cases  $\pi ! i$ )
case ( $Inl \ v$ )

have  $snd3 (nthtriple \ \Pi \ i) = Inl \ v$ 

```

```

apply (unfold snds-of-triples-form-comlist[OF ‹i < chainlen Π›])
apply (auto simp add: Π-def comlist-mk-ps-chain Inl)
done

with wf-Πi wf-ps-triple-def obtain σ where
  v-notin-σ: v ∉ fmdom σ and
  fst-Πi: fst3 (nthtriple Π i) = [ {|v| } | => Top ] ++f σ and
  thd-Πi: thd3 (nthtriple Π i) = [ {|v| } | => Bot ] ++f σ by auto

show prov-triple (asn (ps-to-int G (fst3 (nthtriple Π i))),
  cs ! i, asn (ps-to-int G (thd3 (nthtriple Π i))))
apply (intro prov-vertex[where v=v])
apply (metis (no-types) Inl ‹i < length π› π-cs o-def sum.simps(5))
apply (metis (lifting) Inl lins(2) Inl-not-Inr PlusE ‹i < length π›
  nth-mem sum.simps(1) vertices.simps)
apply (unfold fst-Πi thd-Πi)
apply (unfold ps-to-int-def)
apply (unfold fmdom-add fdom-make-fmap)
apply (unfold finset-is-union[symmetric])
apply (insert v-notin-σ)
apply (unfold iter-hcomp-insert)
apply (unfold lookup-union2 lookup-make-fmap1)
apply (unfold G-def labelling.simps)
apply (subgoal-tac ∀ va ∈ fset (fmdom σ). case-topbot top-ass bot-ass
  (lookup ([ {|v| } | => Top ] ++f σ) va) (Λ va) =
  case-topbot top-ass bot-ass (lookup ([{|v| } | => Bot] ++f σ) va)(Λ va))
apply (unfold iter-hcomp-cong, simp)
apply (metis lookup-union1, simp)
done
next
case (Inr e)
have snd3 (nthtriple Π i) = Inr e
apply (unfold snds-of-triples-form-comlist[OF ‹i < chainlen Π›])
apply (auto simp add: Π-def comlist-mk-ps-chain Inr)
done

with wf-Πi wf-ps-triple-def obtain σ where
  fst-e-disjoint-σ: fst3 e |∩| fmdom σ = {||} and
  thd-e-disjoint-σ: thd3 e |∩| fmdom σ = {||} and
  fst-Πi: fst3 (nthtriple Π i) = [ fst3 e | => Bot ] ++f σ and
  thd-Πi: thd3 (nthtriple Π i) = [ thd3 e | => Top ] ++f σ
by (auto simp add: inf-sup-distrib2)

let ?F = ⊗ v|∈|fmdom σ.
  case-topbot top-ass bot-ass (lookup ([ fst3 e | => Bot ] ++f σ) v) (Λ v)

show prov-triple (asn (ps-to-int G (fst3 (nthtriple Π i))),
  cs ! i, asn (ps-to-int G (thd3 (nthtriple Π i))))

```

```

proof (intro prov-edge[where e=e])
  show e ∈ set E
    apply (subgoal-tac Inr e ∈ set π)
    apply (metis Inr-not-Inl PlusE edges.simps lins(2) sum.simps(2))
    by (metis Inr ‹i < length π› nth-mem)
next
  show coms-com (snd3 e) (cs ! i)
    by (metis (no-types) Inr ‹i < length π› π-cs o-def sum.simps(6))
next
  show ps-to-int G (fst3 (nthtriple Π i)) = ((⊗ v|∈|fst3 e. bot-ass (Λ v)) ⊗
?F)
unfolding fst-Πi ps-to-int-def G-def labelling.simps fmdom-add fdom-make-fmap
  apply (insert fst-e-disjoint-σ)
  apply (unfold iter-hcomp-union)
  apply (subgoal-tac ∀ v ∈ fset (fst3 e). case-topbot top-ass bot-ass
(lookup ([fst3 e |=> Bot] ++f σ) v) (Λ v) = bot-ass (Λ v))
  apply (unfold iter-hcomp-cong)
  apply (simp)
  apply (intro ballI)
  apply (subgoal-tac v |∉| fmdom σ)
  apply (unfold lookup-union2)
  apply (metis lookup-make-fmap topbot.simps(4))
  by (metis fempty-iff finterI)
next
  show ps-to-int G (thd3 (nthtriple Π i)) = ((⊗ v|∈|thd3 e. top-ass (Λ v))
⊗ ?F)
unfolding thd-Πi ps-to-int-def G-def labelling.simps fmdom-add fdom-make-fmap
  apply (insert thd-e-disjoint-σ)
  apply (unfold iter-hcomp-union)
  apply (subgoal-tac ∀ v ∈ fset (thd3 e). case-topbot top-ass bot-ass
(lookup ([thd3 e |=> Top] ++f σ) v) (Λ v) = top-ass (Λ v))
  apply (unfold iter-hcomp-cong)
  apply (subgoal-tac ∀ v ∈ fset (fmdom σ). case-topbot top-ass bot-ass
(lookup ([thd3 e |=> Top] ++f σ) v) (Λ v) =
case-topbot top-ass bot-ass (lookup ([fst3 e |=> Bot] ++f σ) v) (Λ v))
  apply (unfold iter-hcomp-cong)
  apply simp
  subgoal
    by (simp add: lookup-union1)
  subgoal
    by (simp add: fdisjoint-iff lookup-union2 lookup-make-fmap)
  done
qed
qed
qed
qed
qed

```

The soundness theorem states that any diagram provable using the proof

rules for ribbons can be recreated as a valid proof in separation logic.

corollary *soundness-graphical*:

assumes $\wedge c. wr-com\ c = \{\}$

assumes *prov-dia* $G\ P\ Q$

shows $\forall c. coms-dia\ G\ c \longrightarrow prov-triple\ (asn\ P,\ c,\ asn\ Q)$

using *soundness-graphical-helper*[*OF* *assms*(1)] **and** *assms*(2) **by** *auto*

end

References

- [1] J. Bean. *Ribbon Proofs - A Proof System for the Logic of Bunched Implications*. PhD thesis, Queen Mary University of London, 2006.
- [2] R. Bornat, C. Calcagno, and H. Yang. Variables as resource in separation logic. In *Proceedings of the 21st Annual Conference on Mathematical Foundations of Programming Semantics (MFPS XXI)*, volume 155 of *Electronic Notes in Theoretical Computer Science*, pages 247–276. Elsevier, 2006.
- [3] J. Wickerson. *Concurrent Verification for Sequential Programs*. PhD thesis, University of Cambridge, 2013.
- [4] J. Wickerson, M. Dodds, and M. J. Parkinson. Ribbon proofs for separation logic. In M. Felleisen and P. Gardner, editors, *Proceedings of the 22nd European Symposium on Programming (ESOP '13)*, 2013. To appear.