Ribbon Proofs for Separation Logic (Isabelle Formalisation)

John Wickerson

March 17, 2025

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

unbundle *lifting-syntax*
unbundle *fmap.lifting*

type-notation *fmap* (**infix** ‹$\rightharpoonup_f$› *9*)

2.1 Difference

definition
map-diff :: $('k \rightharpoonup 'v) \Rightarrow 'k\ fset \Rightarrow ('k \rightharpoonup 'v)$
where
map-diff f ks = *restrict-map f* (− *fset ks*)

lift-definition
fmap-diff :: $('k \rightharpoonup_f 'v) \Rightarrow 'k\ fset \Rightarrow ('k \rightharpoonup_f 'v)$ (**infix** ‹$\ominus$› *110*)
is *map-diff*
by (*auto simp add*: *map-diff-def*)

2.2 Comprehension

definition
make-map :: $'k\ fset \Rightarrow 'v \Rightarrow ('k \rightharpoonup 'v)$
where
make-map ks v $\equiv \lambda k.$ *if* $k \in$ *fset ks then Some v else None*

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

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

lift-definition
  make-fmap :: $'k$ *fset* ⇒ $'v$ ⇒ ($'k \rightharpoonup_f$ $'v$) (‹[- |=> -]›)
is *make-map* **parametric** *make-map-transfer*
by (*unfold make-map-def dom-def*, *auto*)

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

2.3 Domain

lemma *fmap-add-commute*:
  assumes *fmdom* A |∩| *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* |∪| *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 :: ($'k \rightharpoonup_f$ $'v$) ⇒ $'k$ ⇒ $'v$
is (∘) *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*:
  $\mathit{lookup}\ [\ \{|k|\}\ |{=}{>}\ v\ ]\ k = v$
by (*metis finsert.rep-eq insert-iff lookup-make-fmap*)

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

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

lemma *lookup-union3*:
  assumes $k\ |{\notin}|\ \mathit{fmdom}\ \mathit{xs}$
  shows $\mathit{lookup}\ (\mathit{xs}\ {+}{+}_f\ \mathit{ys})\ k = \mathit{lookup}\ \mathit{ys}\ k$
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*:
  $[\![\ i < \mathit{length}\ \mathit{xs}\ ;\ \mathit{xs}\ !\ i = x\ ]\!] \Longrightarrow x \in \mathit{set}\ \mathit{xs}$ **by** *auto*

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

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

by *auto*

3.1 Projection functions on triples

definition *fst3* :: $'a \times 'b \times 'c \Rightarrow 'a$
where *fst3* $\equiv$ *fst*

definition *snd3* :: $'a \times 'b \times 'c \Rightarrow 'b$
where *snd3* $\equiv$ *fst* $\circ$ *snd*

definition *thd3* :: $'a \times 'b \times 'c \Rightarrow 'c$
where *thd3* $\equiv$ *snd* $\circ$ *snd*

lemma *fst3-simp*:
  $\bigwedge a\ b\ c.\ fst3\ (a,b,c) = a$
by (*auto simp add*: *fst3-def*)

lemma *snd3-simp*:
  $\bigwedge a\ b\ c.\ snd3\ (a,b,c) = b$
by (*auto simp add*: *snd3-def*)

lemma *thd3-simp*:
  $\bigwedge a\ b\ c.\ thd3\ (a,b,c) = c$
by (*auto simp add*: *thd3-def*)

lemma *tripleI*:
  fixes *T U*
  assumes *fst3 T* = *fst3 U*
    and *snd3 T* = *snd3 U*
    and *thd3 T* = *thd3 U*
  shows *T* = *U*
by (*metis assms fst3-def snd3-def thd3-def o-def surjective-pairing*)

end

4 Proof chains

theory *Proofchain* **imports**
  JHelper
begin

An $('a, 'c)$ chain is a sequence of alternating $'a$'s and $'c$'s, beginning and ending with an $'a$. Usually $'a$ represents some sort of assertion, and $'c$ represents some sort of command. Proof chains are useful for stating the SMain proof rule, and for conducting the proof of soundness.

datatype $('a,'c)$ *chain* =
  cNil $'a$ (‹⦃ - ⦄›)
| *cCons* $'a$ $'c$ $('a,'c)$ *chain* (‹⦃ - ⦄ · - · -› [*0*,*0*,*0*] *60*)

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

4.1 Projections

Project first assertion.

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

Project final assertion.

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

Project list of commands.

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

4.2 Chain length

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

lemma *len-comlist-chainlen*:
  $length\ (comlist\ \Pi) = chainlen\ \Pi$
by (*induct* Π, *auto*)

4.3 Extracting triples from chains

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

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 Π i
  shows $i < \mathit{chainlen}\ \Pi \Longrightarrow \mathit{snd3}\ (\mathit{nthtriple}\ \Pi\ i) = (\mathit{comlist}\ \Pi)!i$
apply (*induct* Π *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
  $\mathit{chain\text{-}all} :: (('a \times 'c \times 'a) \Rightarrow \mathit{bool}) \Rightarrow ('a,'c)\ \mathit{chain} \Rightarrow \mathit{bool}$
where
  $\mathit{chain\text{-}all}\ \varphi\ \{\!\mid \sigma \mid\!\} = \mathit{True}$
| $\mathit{chain\text{-}all}\ \varphi\ (\{\!\mid \sigma \mid\!\} \cdot x \cdot \Pi) = (\varphi\ (\sigma,x,\mathit{pre}\ \Pi) \wedge \mathit{chain\text{-}all}\ \varphi\ \Pi)$

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

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

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

4.5 A map function for proof chains

chainmap f g Π 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
$chainmap\ f\ g\ \{\!|\ P\ |\!\} = \{\!|\ f\ P\ |\!\}$
$|\ chainmap\ f\ g\ (\{\!|\ P\ |\!\} \cdot x \cdot \Pi) = \{\!|\ f\ P\ |\!\} \cdot g\ x \cdot chainmap\ f\ g\ \Pi$

Mapping over a chain preserves its length.

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

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

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

lemma *nthtriple-chainmap*:

assumes $i <$ *chainlen* Π
shows *nthtriple* (*chainmap* f g Π) i
$= (\lambda t.\ (f\ (\mathit{fst3}\ t),\ g\ (\mathit{snd3}\ t),\ f\ (\mathit{thd3}\ t)))\ (\mathit{nthtriple}\ \Pi\ i)$
using *assms*
proof (*induct* Π *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 :: $('a,'c)$ *chain* $\Rightarrow$ $'c \Rightarrow$ $'a \Rightarrow$ $('a,'c)$ *chain*
where
cSnoc $\{\!|\ \sigma\ |\!\}\ y\ \tau = \{\!|\ \sigma\ |\!\} \cdot y \cdot \{\!|\ \tau\ |\!\}$
| *cSnoc* $(\{\!|\ \sigma\ |\!\} \cdot x \cdot \Pi)\ y\ \tau = \{\!|\ \sigma\ |\!\} \cdot x \cdot (\mathit{cSnoc}\ \Pi\ y\ \tau)$

lemma *len-snoc*:
fixes Π x P
shows *chainlen* (*cSnoc* Π x P) $= 1 +$ (*chainlen* Π)
by (*induct* Π, *auto*)

lemma *pre-snoc*:
pre (*cSnoc* Π x P) $=$ *pre* Π
by (*induct* Π, *auto*)

lemma *post-snoc*:
post (*cSnoc* Π x P) $= P$
by (*induct* Π, *auto*)

lemma *comlist-snoc*:
comlist (*cSnoc* Π x p) $=$ *comlist* Π @ $[x]$
by (*induct* Π, *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* ⇒ *assertion* ⇒ *assertion* (**infixr** ‹⋆› *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: $p \star Emp = p$ **and**
  emp-star: $Emp \star p = p$

lemma *star-rot*:
  $q \star p \star r = p \star q \star r$
using *star-assoc star-comm* **by** *auto*

axiomatization
  Exists :: *string* ⇒ *assertion* ⇒ *assertion*

Extracting the set of program variables mentioned in an assertion.

axiomatization
  rd-ass :: *assertion* ⇒ *string set*
where *rd-emp*: *rd-ass Emp* = {}
  and *rd-star*: *rd-ass* $(p \star q)$ = *rd-ass p* ∪ *rd-ass q*
  and *rd-exists*: *rd-ass* (*Exists x p*) = *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* ⇒ *command* ⇒ *command*

axiomatization
  Loop :: *command* ⇒ *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** ‹$\otimes_c$› *50*)
| *Emp-int-conc* (‹ε_c›)
| *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* ⇒ *conc-interface* ⇒ *bool* (**infix** ‹≃› *45*)
where
  refl: $P \simeq P$
| *sym*: $P \simeq Q \Longrightarrow Q \simeq P$

| *trans*: $[\![P \simeq Q;\ Q \simeq R]\!] \Longrightarrow P \simeq R$
| *cong-hcomp1*: $P \simeq Q \Longrightarrow P' \otimes_c P \simeq P' \otimes_c Q$
| *cong-hcomp2*: $P \simeq Q \Longrightarrow P \otimes_c P' \simeq Q \otimes_c P'$
| *cong-exists*: $P \simeq Q \Longrightarrow$ *Exists-int-conc* x $P \simeq$ *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*:
$[\![\ P \simeq Q\ ;\ P' \simeq Q'\ ]\!] \Longrightarrow P \otimes_c P' \simeq Q \otimes_c Q'$
by (*metis cong-hcomp1 cong-hcomp2 equiv-int.trans*)

quotient-type *interface* = *conc-interface* / *equiv-int*
apply (*intro equivpI*)
apply (*intro reflpI*, *simp add*: *equiv-int.refl*)
apply (*intro sympI*, *simp add*: *equiv-int.sym*)
apply (*intro transpI*, *elim equiv-int.trans*, *simp add*: *equiv-int.refl*)
done

lift-definition
Ribbon :: *assertion* ⇒ *interface*
is *Ribbon-conc* **.**

lift-definition
Emp-int :: *interface* (‹ε›)
is ε_c **.**

lift-definition
Exists-int :: *string* ⇒ *interface* ⇒ *interface*
is *Exists-int-conc*
by (*rule equiv-int.cong-exists*)

lift-definition
HComp-int :: *interface* ⇒ *interface* ⇒ *interface* (**infix** ‹⊗› *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*:
  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
  iter-hcomp $X\ f \equiv$ *ffold* $((\otimes) \circ f)\ \varepsilon\ X$

syntax *iter-hcomp-syntax* ::
  $'a \Rightarrow ('a\ fset) \Rightarrow ('a \Rightarrow interface) \Rightarrow interface$
    (‹($\bigotimes$ -|∈|-. -)› [*0*,*0*,*10*] *10*)
syntax-consts *iter-hcomp-syntax* == *iter-hcomp*
translations $\bigotimes x|\in|M.\ e$ == *CONST iter-hcomp* $M\ (\lambda x.\ e)$

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

lemma *iter-hcomp-cong*:
  assumes $\forall\, v \in$ *fset* $vs.\ \varphi\ v = \varphi'\ v$
  shows $(\bigotimes v|\in|vs.\ \varphi\ v) = (\bigotimes v|\in|vs.\ \varphi'\ v)$
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*:
  shows $(\bigotimes x\ |\in|\ \{||\}.\ p\ x) = \varepsilon$
  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*:
  assumes $v\ |\notin|\ ws$
  shows $(\bigotimes x\ |\in|$ *finsert* $v\ ws.\ p\ x) = (p\ v \otimes (\bigotimes x\ |\in|\ ws.\ p\ x))$
proof −
  interpret *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*:
  assumes $vs\ |\cap|\ ws = \{||\}$
  shows $(\bigotimes x\ |\in|\ vs\ |\cup|\ ws.\ p\ x) = ((\bigotimes x\ |\in|\ vs.\ p\ x) \otimes (\bigotimes x\ |\in|\ ws.\ p\ x))$

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* ⇒ *assertion*
where
  conc-asn (*Ribbon-conc p*) = *p*
| *conc-asn* ($P \otimes_c Q$) = (*conc-asn P*) ⋆ (*conc-asn Q*)
| *conc-asn* (ε_c) = *Emp*
| *conc-asn* (*Exists-int-conc x P*) = *Exists x* (*conc-asn P*)

lift-definition
  asn :: *interface* ⇒ *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*) ⋆ (*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* ⇒ *string set*
where
  rd-conc-int (*Ribbon-conc p*) = *rd-ass p*
| *rd-conc-int* ($P \otimes_c Q$) = *rd-conc-int P* ∪ *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* ⇒ *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*:
$\bigwedge\Pi.$ $[\![$ *length cs* = *chainlen* Π ; *p1* = *asn* (*pre* Π) ; *p2* = *asn* (*post* Π) ;
$\bigwedge i.$ $i <$ *chainlen* $\Pi \Longrightarrow$ *prov-triple*
(*asn* (*fst3* (*nthtriple* Π *i*)), *cs* ! *i*, *asn* (*thd3* (*nthtriple* Π *i*))) $]\!]$
$\Longrightarrow$ *prov-triple* (*p1*, *foldr* (;;) *cs 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* Π' **where** Π-*def*: $\Pi = \{\!|\ p\ |\!\} \cdot x \cdot \Pi'$
by (*metis Cons.prems*(*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.prems*(*2*) *Cons.prems*(*3*) Π-*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* Π-*def*, *intro Cons.prems*(*4*), *simp add*: Π-*def*)
apply (*intro Cons.hyps*, *insert* Π-*def Cons.prems*(*1*), *auto*)
apply (*fold nth-Cons-Suc*[*of c cs*] *nthtriple.simps*(*2*)[*of p x* Π'])
apply (*fold* Π-*def*, *intro Cons.prems*(*4*), *simp add*: Π-*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* × *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* ϱs) = *foldr* (;;) (*map* (*com-cell* ∘ *fst*) ϱs) *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* ϱs) = ($\bigcup$ *r* ∈ *set* ϱs. *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 ϱs :: *row list*
  and ϱ :: *row*
  shows (*wr-sdia D* = *wr-com* (*com-sdia D*))
  and (*wr-cell* γ = *wr-com* (*com-cell* γ))
  and ($\bigcup$ ϱ ∈ *set* ϱs. *wr-cell* (*fst* ϱ))
    = *wr-com* (*foldr* (;;) (*map* ($\lambda(\gamma,F)$. *com-cell* γ) ϱs) *Skip*)
  and *wr-cell* (*fst* ϱ) = *wr-com* (*com-cell* (*fst* ϱ))
apply (*induct D* **and** γ **and** ϱs **and** ϱ *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-sdia :: [*sdiagram*, *interface*, *interface*] $\Rightarrow$ *bool* **and**
  prov-row :: [*row*, *interface*, *interface*] $\Rightarrow$ *bool* **and**
  prov-cell :: [*cell*, *interface*, *interface*] $\Rightarrow$ *bool*
where
  SRibbon: *prov-cell* (*Filler P*) *P P*
| *SBasic*: *prov-triple* (*asn P*, *c*, *asn Q*) $\Longrightarrow$ *prov-cell* (*Basic P c Q*) *P Q*
| *SExists*: *prov-sdia D P Q*
  $\Longrightarrow$ *prov-cell* (*Exists-sdia x D*) (*Exists-int x P*) (*Exists-int x Q*)
| *SChoice*: $\llbracket$ *prov-sdia D P Q* ; *prov-sdia E P Q* $\rrbracket$
  $\Longrightarrow$ *prov-cell* (*Choose-sdia P D E Q*) *P Q*
| *SLoop*: *prov-sdia D P P* $\Longrightarrow$ *prov-cell* (*Loop-sdia P D P*) *P P*
| *SRow*: $\llbracket$ *prov-cell* γ *P Q* ; *wr-cell* $\gamma \cap$ *rd-int F* $= \{\}$ $\rrbracket$
  $\Longrightarrow$ *prov-row* (γ, *F*) ($P \otimes F$) ($Q \otimes F$)
| *SMain*: $\llbracket$ *chain-all* ($\lambda(P,\varrho,Q)$. *prov-row* ϱ *P Q*) Π ; *0* $<$ *chainlen* Π $\rrbracket$
  $\Longrightarrow$ *prov-sdia* (*SDiagram* (*comlist* Π)) (*pre* Π) (*post* Π)

7.3 Soundness

lemma *soundness-strat-helper*:
  (*prov-sdia D P Q* $\longrightarrow$ *prov-triple* (*asn P*, *com-sdia D*, *asn Q*)) $\wedge$
  (*prov-row* ϱ *P Q* $\longrightarrow$ *prov-triple* (*asn P*, *com-cell* (*fst* ϱ), *asn Q*)) $\wedge$
  (*prov-cell* γ *P Q* $\longrightarrow$ *prov-triple* (*asn P*, *com-cell* γ, *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* γ *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* Π)
  thus *?case*
  apply (*unfold com-sdia.simps*)
  apply (*intro seq-fold*[*of* - Π])
  apply (*simp-all add*: *len-comlist-chainlen*)[*3*]

```
  apply (induct Π, 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 ⇒ assertion-gadget
  (node fset × command-gadget × node fset) list
type-synonym labelling = node ⇒ assertion-gadget
type-synonym edge = node fset × command-gadget × node fset
```

Projecting components from a graph

```
fun vertices :: diagram ⇒ node fset (‹-^V› [1000] 1000)
where (Graph V Λ E)^V = V
```

term *this* $(is\hat{\ }V) = (a\ test)\hat{\ }V$

fun *labelling* :: *diagram* ⇒ *labelling* (‹-^Λ› [*1000*] *1000*)
where (*Graph V* Λ *E*)^Λ = Λ

fun *edges* :: *diagram* ⇒ *edge list* (‹-^*E*› [*1000*] *1000*)
where (*Graph V* Λ *E*)^*E* = *E*

8.2 Well formedness of graphical diagrams

definition *acyclicity* :: *edge list* ⇒ *bool*
where
  acyclicity E ≡ *acyclic* (⋃ *e* ∈ *set E*. *fset* (*fst3 e*) × *fset* (*thd3 e*))

definition *linearity* :: *edge list* ⇒ *bool*
where
  linearity E ≡
    distinct E ∧ (∀ *e* ∈ *set E*. ∀ *f* ∈ *set E*. *e* ≠ *f* ⟶
    fst3 e |∩| *fst3 f* = {||} ∧
    thd3 e |∩| *thd3 f* = {||})

lemma *linearityD*:
  assumes *linearity E*
  shows *distinct E*
  and ⋀*e f*. ⟦ *e* ∈ *set E* ; *f* ∈ *set E* ; *e* ≠ *f* ⟧ ⟹
    fst3 e |∩| *fst3 f* = {||} ∧
    thd3 e |∩| *thd3 f* = {||}
using *assms* **unfolding** *linearity-def* **by** *auto*

lemma *linearityD2*:
  linearity E ⟹ (∀ *e f*. *e* ∈ *set E* ∧ *f* ∈ *set E* ∧ *e* ≠ *f* ⟶
    fst3 e |∩| *fst3 f* = {||} ∧
    thd3 e |∩| *thd3 f* = {||})
unfolding *linearity-def* **by** *auto*

inductive
  wf-ass :: *assertion-gadget* ⇒ *bool* **and**
  wf-com :: *command-gadget* ⇒ *bool* **and**
  wf-dia :: *diagram* ⇒ *bool*
where
  wf-rib: *wf-ass* (*Rib p*)
| *wf-exists*: *wf-dia G* ⟹ *wf-ass* (*Exists-dia x G*)
| *wf-com*: *wf-com* (*Com c*)
| *wf-choice*: ⟦ *wf-dia G* ; *wf-dia H* ⟧ ⟹ *wf-com* (*Choose-dia G H*)
| *wf-loop*: *wf-dia G* ⟹ *wf-com* (*Loop-dia G*)
| *wf-dia*: ⟦ ∀ *e* ∈ *set E*. *wf-com* (*snd3 e*) ; ∀ *v* ∈ *fset V*. *wf-ass* (Λ *v*) ;
  acyclicity E ; *linearity E* ; ∀ *e* ∈ *set E*. *fst3 e* |∪| *thd3 e* |⊆| *V* ⟧ ⟹
  wf-dia (*Graph V* Λ *E*)

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

lemma *wf-dia-inv*:
  assumes *wf-dia* (*Graph* V Λ E)
  shows $\forall$ $v \in$ *fset* V. *wf-ass* (Λ v)
    and $\forall$ $e \in$ *set* E. *wf-com* (*snd3* e)
    and *acyclicity* E
    and *linearity* E
    and $\forall$ $e \in$ *set* E. *fst3* e $|\cup|$ *thd3* e $|\subseteq|$ 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* (λv. ($\forall$ $e \in$ *set* $G\hat{}E$. v $|\notin|$ *thd3* e)) $G\hat{}V$

definition
  terminals :: *diagram* $\Rightarrow$ *node fset*
where
  terminals G = *ffilter* (λv. ($\forall$ $e \in$ *set* $G\hat{}E$. v $|\notin|$ *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 $|\subseteq|$ $G\hat{}V$
unfolding *initials-def* **by** *auto*

lemma *terminals-in-vertices*:
  terminals G $|\subseteq|$ $G\hat{}V$
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) = ($\bigotimes$ v $|\in|$ *initials* (*Graph* V Λ E). *top-ass* (Λ v))

| *top-ass* (*Rib p*) = *Ribbon p*
| *top-ass* (*Exists-dia x G*) = *Exists-int x* (*top-dia G*)

primrec
  bot-ass :: *assertion-gadget* ⇒ *interface* **and**
  bot-dia :: *diagram* ⇒ *interface*
where
  bot-dia (*Graph V* Λ *E*) = (⨂ *v* |∈| *terminals* (*Graph V* Λ *E*). *bot-ass* (Λ *v*))
| *bot-ass* (*Rib p*) = *Ribbon p*
| *bot-ass* (*Exists-dia x G*) = *Exists-int x* (*bot-dia G*)

8.5 Proof rules for graphical diagrams

inductive
  prov-dia :: [*diagram*, *interface*, *interface*] ⇒ *bool* **and**
  prov-com :: [*command-gadget*, *interface*, *interface*] ⇒ *bool* **and**
  prov-ass :: *assertion-gadget* ⇒ *bool*
where
  Skip: *prov-ass* (*Rib p*)
| *Exists*: *prov-dia G* - - ⟹ *prov-ass* (*Exists-dia x G*)
| *Basic*: *prov-triple* (*asn P*, *c*, *asn Q*) ⟹ *prov-com* (*Com c*) *P Q*
| *Choice*: ⟦ *prov-dia G P Q* ; *prov-dia H P Q* ⟧
    ⟹ *prov-com* (*Choose-dia G H*) *P Q*
| *Loop*: *prov-dia G P P* ⟹ *prov-com* (*Loop-dia G*) *P P*
| *Main*: ⟦ *wf-dia G* ; ⋀*v*. *v* ∈ *fset G^V* ⟹ *prov-ass* (*G^Λ v*);
    ⋀*e*. *e* ∈ *set G^E* ⟹ *prov-com* (*snd3 e*)
      (⨂ *v* |∈| *fst3 e*. *bot-ass* (*G^Λ v*))
      (⨂ *v* |∈| *thd3 e*. *top-ass* (*G^Λ v*))⟧
    ⟹ *prov-dia G* (*top-dia G*) (*bot-dia G*)

inductive-cases *main-inv*: *prov-dia* (*Graph V* Λ *E*) *P Q*
inductive-cases *loop-inv*: *prov-com* (*Loop-dia G*) *P Q*
inductive-cases *choice-inv*: *prov-com* (*Choose-dia G H*) *P Q*
inductive-cases *basic-inv*: *prov-com* (*Com c*) *P Q*
inductive-cases *exists-inv*: *prov-ass* (*Exists-dia x G*)
inductive-cases *skip-inv*: *prov-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* ⇒ *lin set*
where
lins G ≡ {π :: *lin*.
    (*distinct* π)

$\wedge$ (*set* π = (*fset* G^V) <+> (*set* G^E))
$\wedge$ ($\forall$ *i j v e*. *i* < *length* π $\wedge$ *j* < *length* π $\wedge$ π!*i* = *Inl v* $\wedge$ π!*j* = *Inr e*
$\wedge$ *v* |$\in$| *fst3 e* $\longrightarrow$ *i*<*j*)
$\wedge$ ($\forall$ *j k w e*. *j* < *length* π $\wedge$ *k* < *length* π $\wedge$ π!*j* = *Inr e* $\wedge$ π!*k* = *Inl w*
$\wedge$ *w* |$\in$| *thd3 e* $\longrightarrow$ *j*<*k*) }

lemma *linsD*:
assumes $\pi \in$ *lins G*
shows (*distinct* π)
and (*set* π = (*fset* G^V) <+> (*set* G^E))
and ($\forall$ *i j v e*. *i* < *length* π $\wedge$ *j* < *length* π
$\wedge$ π!*i* = *Inl v* $\wedge$ π!*j* = *Inr e* $\wedge$ *v* |$\in$| *fst3 e* $\longrightarrow$ *i*<*j*)
and ($\forall$ *j k w e*. *j* < *length* π $\wedge$ *k* < *length* π
$\wedge$ π!*j* = *Inr e* $\wedge$ π!*k* = *Inl w* $\wedge$ *w* |$\in$| *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*:
($\forall$ *i*<*length* π. *case-sum* (*coms-ass* $\circ$ Λ) (*coms-com* $\circ$ *snd3*) (π!*i*) (*cs*!*i*))
=
(($\forall$ *i*. *i*<*length* π $\wedge$ ($\exists$ *v*. (π!*i*) = *Inl v*) $\longrightarrow$
coms-ass (Λ (*projl* (π!*i*))) (*cs*!*i*)) $\wedge$
($\forall$ *i*. *i*<*length* π $\wedge$ ($\exists$ *e*. (π!*i*) = *Inr e*) $\longrightarrow$
coms-com (*snd3* (*projr* (π!*i*))) (*cs*!*i*)))
apply (*intro iffI*)
apply *auto*[*1*]
apply (*intro allI impI*, *case-tac* π!*i*, *auto*)
done

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*: $\llbracket$ *coms-dia G c*; *coms-dia H d* $\rrbracket \Longrightarrow$
coms-com (*Choose-dia G H*) (*Choose c d*)
| *coms-loop*: *coms-dia G c* $\Longrightarrow$ *coms-com* (*Loop-dia G*) (*Loop c*)
| *coms-main*: $\llbracket$ $\pi \in$ *lins* (*Graph V* Λ *E*); *length cs* = *length* π;
$\forall$ *i*<*length* π. *case-sum* (*coms-ass* $\circ$ Λ) (*coms-com* $\circ$ *snd3*) (π!*i*) (*cs*!*i*) $\rrbracket$
$\Longrightarrow$ *coms-dia* (*Graph V* Λ *E*) (*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* $\rightharpoonup_f$ *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* ⇒ *node* + *edge* ⇒ *proofstate*
where
  next-ps σ (*Inl v*) = σ ⊖ {|*v*|} $++_f$ [{|*v*|} |=> *Bot*]
| *next-ps* σ (*Inr e*) = σ ⊖ *fst3 e* $++_f$ [*thd3 e* |=> *Top*]

The function *mk-ps-chain* Π π 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*] ⇒ *ps-chain*
where
  mk-ps-chain ≡ *foldl* (λΠ *x*. *cSnoc* Π *x* (*next-ps* (*post* Π) *x*))

lemma *mk-ps-chain-preserves-length*:
  fixes π Π
  shows *chainlen* (*mk-ps-chain* Π π) = *chainlen* Π + *length* π
proof (*induct* π *arbitrary*: Π)
  case *Nil*
  show *?case* **by** (*unfold mk-ps-chain-def*, *auto*)
next
  case (*Cons x* π)
  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 Π (*x* # π) = *mk-ps-chain* (*cSnoc* Π *x* (*next-ps* (*post* Π) *x*)) π
by (*auto simp add*: *mk-ps-chain-def*)

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

lemma *mk-ps-chain-snoc*:
  mk-ps-chain Π $(\pi$ @ $[x])$
    $=$ *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* ($\{\!|$ σ $|\!\}$ $\cdot$ x $\cdot$ Π) π $=$ $\{\!|$ σ $|\!\}$ $\cdot$ 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.Ribbons-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 $\mathit{lins2} :: [\mathit{node\ fset},\ \mathit{diagram}] \Rightarrow \mathit{lin\ set}$
where
$\mathit{lins2}\ S\ G \equiv \{\pi :: \mathit{lin}\ .$
$(\mathit{distinct}\ \pi)$
$\wedge\ (\mathit{set}\ \pi = (\mathit{fset}\ G\hat{\ }V - \mathit{fset}\ S) <\!+\!> \mathit{set}\ G\hat{\ }E)$
$\wedge\ (\forall\, i\ j\ v\ e.\ i < \mathit{length}\ \pi \wedge j < \mathit{length}\ \pi$
$\wedge\ \pi!i = \mathit{Inl}\ v \wedge \pi!j = \mathit{Inr}\ e \wedge v\ |{\in}|\ \mathit{fst3}\ e \longrightarrow i{<}j)$
$\wedge\ (\forall\, j\ k\ w\ e.\ j < \mathit{length}\ \pi \wedge k < \mathit{length}\ \pi$
$\wedge\ \pi!j = \mathit{Inr}\ e \wedge \pi!k = \mathit{Inl}\ w \wedge w\ |{\in}|\ \mathit{thd3}\ e \longrightarrow j{<}k)\ \}$

lemma *lins2D*:
assumes $\pi \in \mathit{lins2}\ S\ G$
shows $\mathit{distinct}\ \pi$
and $\mathit{set}\ \pi = (\mathit{fset}\ G\hat{\ }V - \mathit{fset}\ S) <\!+\!> \mathit{set}\ G\hat{\ }E$
and $\bigwedge i\ j\ v\ e.\ [\![\ i < \mathit{length}\ \pi\ ;\ j < \mathit{length}\ \pi\ ;$
$\pi!i = \mathit{Inl}\ v\ ;\ \pi!j = \mathit{Inr}\ e\ ;\ v\ |{\in}|\ \mathit{fst3}\ e\ ]\!] \Longrightarrow i{<}j$
and $\bigwedge i\ k\ w\ e.\ [\![\ j < \mathit{length}\ \pi\ ;\ k < \mathit{length}\ \pi\ ;$
$\pi!j = \mathit{Inr}\ e\ ;\ \pi!k = \mathit{Inl}\ w\ ;\ w\ |{\in}|\ \mathit{thd3}\ e\ ]\!] \Longrightarrow j{<}k$
using *assms*
apply (*unfold lins2-def Collect-iff*)
apply (*elim conjE*, *assumption*)+
apply *blast*+
done

lemma *lins2I*:
assumes $\mathit{distinct}\ \pi$
and $\mathit{set}\ \pi = (\mathit{fset}\ G\hat{\ }V - \mathit{fset}\ S) <\!+\!> \mathit{set}\ G\hat{\ }E$
and $\bigwedge i\ j\ v\ e.\ [\![\ i < \mathit{length}\ \pi\ ;\ j < \mathit{length}\ \pi\ ;$
$\pi!i = \mathit{Inl}\ v\ ;\ \pi!j = \mathit{Inr}\ e\ ;\ v\ |{\in}|\ \mathit{fst3}\ e\ ]\!] \Longrightarrow i{<}j$
and $\bigwedge j\ k\ w\ e.\ [\![\ j < \mathit{length}\ \pi\ ;\ k < \mathit{length}\ \pi\ ;$
$\pi!j = \mathit{Inr}\ e\ ;\ \pi!k = \mathit{Inl}\ w\ ;\ w\ |{\in}|\ \mathit{thd3}\ e\ ]\!] \Longrightarrow j{<}k$
shows $\pi \in \mathit{lins2}\ S\ G$
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*:
$\mathit{lins}\ G = \mathit{lins2}\ \{||\}\ G$
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
$\mathit{initial\text{-}ps} :: \mathit{diagram} \Rightarrow \mathit{proofstate}$

where
initial-ps $G \equiv$ [*initials* G |=> *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 |=> *Top*] $++_f$ [S |=> *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 (σ $++_f$ *disjoint* [{|v|} |=> *Top*], *Inl* v, σ $++_f$ *disjoint* [{|v|} |=> *Bot*]), where v is a node, or of the form (σ $++_f$ *disjoint* [{|vs|} |=> *Bot*

], *Inr e*, σ $++_f$ *disjoint* [{|*ws*|} |=> *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-ps-triple :: *proofstate* × (*node* + *edge*) × *proofstate* ⇒ *bool*
where
  wf-ps-triple T = (*case snd3 T of*
    Inl v ⇒ (∃σ. *v* |∉| *fmdom* σ
      ∧ *fst3 T* = [{|*v*|} |=> *Top*] $++_f$ σ
      ∧ *thd3 T* = [{|*v*|} |=> *Bot*] $++_f$ σ)
  | *Inr e* ⇒ (∃σ. (*fst3 e* |∪| *thd3 e*) |∩| *fmdom* σ = {||}
      ∧ *fst3 T* = [*fst3 e* |=> *Bot*] $++_f$ σ
      ∧ *thd3 T* = [*thd3 e* |=> *Top*] $++_f$ σ))

lemma *wf-ps-triple-nodeI*:
  assumes ∃σ. *v* |∉| *fmdom* σ ∧
    σ*1* = [{|*v*|} |=> *Top*] $++_f$ σ ∧
    σ*2* = [{|*v*|} |=> *Bot*] $++_f$ σ
  shows *wf-ps-triple* (σ*1*, *Inl v*, σ*2*)
using *assms* **unfolding** *wf-ps-triple-def*
by (*auto simp add*: *fst3-simp snd3-simp thd3-simp*)

lemma *wf-ps-triple-edgeI*:
  assumes ∃σ. (*fst3 e* |∪| *thd3 e*) |∩| *fmdom* σ = {||}
      ∧ σ*1* = [*fst3 e* |=> *Bot*] $++_f$ σ
      ∧ σ*2* = [*thd3 e* |=> *Top*] $++_f$ σ
  shows *wf-ps-triple* (σ*1*, *Inr e*, σ*2*)
using *assms* **unfolding** *wf-ps-triple-def*
by (*auto simp add*: *fst3-simp snd3-simp thd3-simp*)

definition
  wf-ps-chain :: *ps-chain* ⇒ *bool*
where
  wf-ps-chain ≡ *chain-all wf-ps-triple*

lemma *next-initial-ps2-vertex*:
  initial-ps2 ({|*v*|} |∪| *S*) *G*
  = *initial-ps2 S G* ⊖ {|*v*|} $++_f$ [{|*v*|} |=> *Bot*]
apply (*unfold initial-ps2-def*)
apply *transfer*
apply (*auto simp add*: *make-map-def map-diff-def map-add-def restrict-map-def*)
done

lemma *next-initial-ps2-edge*:
  assumes *G* = *Graph V* Λ *E* **and** *G′* = *Graph V′* Λ *E′* **and**
    V′ = *V* − *fst3 e* **and** *E′* = *removeAll e E* **and** *e* ∈ *set E* **and**
    fst3 e |⊆| *S* **and** *S* |⊆| *initials G* **and** *wf-dia G*

shows *initial-ps2* $(S - \mathit{fst3}\ e)\ G' =$
initial-ps2 $S\ G \ominus \mathit{fst3}\ e\ {+}{+}_f$ [*thd3 e* |=> *Top*]
proof (*insert assms*, *unfold initial-ps2-def*, *transfer*)
fix $G\ V\ \Lambda\ E\ G'\ V'\ E'\ e\ S$
assume *G-def*: $G = \mathit{Graph}\ V\ \Lambda\ E$ **and** *G′-def*: $G' = \mathit{Graph}\ V'\ \Lambda\ E'$ **and**
V′-def: $V' = V - \mathit{fst3}\ e$ **and** *E′-def*: $E' = \mathit{removeAll}\ e\ E$ **and**
e-in-E: $e \in \mathit{set}\ E$ **and** *fst-e-in-S*: *fst3 e* $|\subseteq|$ *S* **and**
S-initials: *S* $|\subseteq|$ *initials G* **and** *wf-G*: *wf-dia G*
have *thd3 e* $|\cap|$ *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* $|\cap|$ *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 *Inr e # π ∈ lins2 S (Graph V Λ E)*
    and *vs |⊆| S*
    and *e = (vs,c,ws)*
  shows *π ∈ lins2 (S − vs) (Graph (V − vs) Λ (removeAll e E))*
proof −
  note *lins2D = lins2D[OF assms(1)]*
  show *?thesis*
  proof (*intro lins2I*, *unfold vertices.simps edges.simps*)
    show *distinct π*
    using *lins2D(1)* **by** *auto*
  next
    show *set π = (fset (V − vs) − fset (S − vs))*
      <+> set (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 ∀ *a aa b.*
        insert (Inr (vs, c, ws)) (set π) = (fset V − fset S) <+> set E ⟶
        fset vs ⊆ fset S ⟶
        Inr (vs, c, ws) ∉ set π ⟶
        distinct π ⟶ (a, aa, b) ∈ set E ⟶ Inr (a, aa, b) ∉ set π ⟶ b = ws
      by (*metis (lifting) InrI List.set-simps(2)*
      prod.inject set-ConsD sum.simps(2))

      moreover have ∀ *a aa b.*
        insert (Inr (vs, c, ws)) (set π) = (fset V − fset S) <+> set E ⟶
        fset vs ⊆ fset S ⟶
        Inr (vs, c, ws) ∉ set π ⟶
        distinct π ⟶ (a, aa, b) ∈ set E ⟶ Inr (a, aa, b) ∉ set π ⟶ aa = c
      by (*metis (lifting) InrI List.set-simps(2)*
      prod.inject set-ConsD sum.simps(2))

      moreover have ∀ *x. insert (Inr (vs, c, ws)) (set π) = (fset V − fset S) <+>
set E ⟶*
          fset vs ⊆ fset S ⟶
          Inr (vs, c, ws) ∉ set π ⟶
          *distinct π ⟶ x ∈ set π ⟶ x ∈ (fset V − fset S) <+> set E − {(vs, c,
ws)}*
        apply (*unfold insert-is-Un[of - set π]*)
        apply (*fold assms(3)*)
        apply *clarify*
        apply (*subgoal-tac set π = ((fset V − fset S) <+> set E) − {Inr e}*)
        by *auto*
    ultimately show *Inr (vs, c, ws) ∉ set π ∧ distinct π ⟹*
      insert (Inr (vs, c, ws)) (set π) = (fset V − fset S) <+> set E ⟹

fset vs $\subseteq$ *fset S* $\Longrightarrow$ *set* π = (*fset V* $-$ *fset S*) <+> *set E* $-$ $\{$(*vs*, *c*, *ws*)$\}$
by *blast*
qed
next
fix *i j v e*
assume *i* $<$ *length* π *j* $<$ *length* π π ! *i* = *Inl v*
π ! *j* = *Inr e v* $|\in|$ *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* $|\in|$ *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* $|\subseteq|$ *initials G*
and *wf-dia G*
and $\Pi \in$ *ps-chains2 S G*
and *fcard G^V* + *length G^E* = *k* + *fcard S*
shows *wf-ps-chain* Π $\wedge$ (*post* Π = [*terminals G* |=> *Bot*])
using *assms*
proof (*induct k arbitrary*: *S G* Π)
case *0*
obtain *V* Λ *E* **where** *G-def*: *G* = *Graph V* Λ *E* **by** (*metis diagram.exhaust*)
have *S* $|\subseteq|$ *V*
using *0.prems*(*1*) *initials-in-vertices*[*of G*]
by (*auto simp add*: *G-def*)
have *fcard V* $\leq$ *fcard S*
using *0.prems*(*4*)
by (*unfold G-def*, *auto*)
from *fcard-seteq*[*OF* ‹*S* $|\subseteq|$ *V*› *this*] **have** *S* = *V* **by** *auto*
hence *E* = [] **using** *0.prems*(*4*) **by** (*unfold G-def*, *auto*)
have *initials G* = *V*
by (*unfold G-def* ‹*E*=[]›, *rule no-edges-imp-all-nodes-initial*)
have *terminals G* = *V*
by (*unfold G-def* ‹*E*=[]›, *rule no-edges-imp-all-nodes-terminal*)
have {} <+> {} = {} **by** *auto*

```
  have lins2 S G = { [] }
  apply (unfold G-def ‹S=V› ‹E=[]›)
  apply (unfold lins2-def, auto simp add: ‹{} <+> {} = {}›)
  done
  hence Π-def: Π = ⦃ 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 Π-def wf-ps-chain-def, auto)
  apply (unfold post.simps initial-ps2-def ‹initials G = V› ‹terminals G = V›)
  apply (unfold ‹S=V›)
  apply (subgoal-tac V − V = {||}, simp-all)
  done
next
  case (Suc k)
  obtain V Λ E where G-def: G = Graph V Λ E by (metis diagram.exhaust)
  from Suc.prems(3) obtain π where
    Π-def: Π = mk-ps-chain ⦃ initial-ps2 S G ⦄ π and
    π-in: π ∈ lins2 S G
    by (auto simp add: ps-chains2-def)
  note lins2 = lins2D[OF π-in]
  have S |⊆| V
    using Suc.prems(1) initials-in-vertices[of G]
    by (auto simp add: G-def)
  show ?case
  proof (cases π)
    case Nil
    from π-in have V = S E = []
    apply (−, unfold ‹π = []› lins2-def, simp-all)
    apply (unfold empty-eq-Plus-conv)
    apply (unfold G-def vertices.simps edges.simps, auto)
    by (metis ‹S |⊆| V› 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 π′)
    note π-def = this
    show ?thesis
    proof (cases x)
      case (Inl v)
      note x-def = this

      have v |∉| S ∧ v |∈| V
      apply (subgoal-tac v ∈ fset V − fset S)
      apply (simp)
      apply (subgoal-tac Inl v ∈ (fset V − fset S) <+> 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.prems(1) fset-rev-mp)
next
  show wf-dia G by (rule Suc.prems(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* π*-def*)
apply (*rule* π*-in*)
by (*metis v-notin-S*)
next
show *fcard G^V + length G^E = k + fcard S′*
apply (*unfold S′-def*)
by (*auto simp add*: *Suc.prems*(*4*) *fcard-finsert-disjoint*[*OF v-notin-S*])
qed
hence
wf-Π′: *wf-ps-chain* Π′ **and**
post-Π′: *post* Π′ = [*terminals G* |=> *Bot*]
by *auto*

show *?thesis*
proof (*intro conjI*)
have *1*: *fmdom* [{|*v*|} |=> *Bot*]
|∩| *fmdom* ([*initials G* − ({|*v*|} |∪| *S*) |=> *Top*] $++_f$
[*S* |=> *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* Π
using [[*unfold-abs-def* = *false*]]
apply (*simp only*: Π*-def* π*-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* Π′*-def*)
apply (*unfold wf-ps-chain-def chain-all.simps conj-commute*)
apply (*intro conjI*)
apply (*fold wf-ps-chain-def*, *rule wf*-Π′)
apply (*intro wf-ps-triple-nodeI exI*[*of -* σ] *conjI*)
apply (*unfold* σ*-def fmdom-add fdom-make-fmap*)
apply (*metis finsertI1 fminus-iff funion-iff v-notin-S*)
apply (*unfold pre*-Π′ *initial-ps2-def S′-def*)
apply (*unfold fmap-add-commute*[*OF 1*])
apply (*unfold fmadd-assoc*)
apply (*fold fmadd-assoc*[*of -* [*S* |=> *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* Π = [*terminals G* |=> *Bot*]
apply (*unfold* Π*-def* π*-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 Π′-def, rule post-Π′)
    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: ⋀e. e ∈ set E ⟹ fst3 e |∪| thd3 e |⊆| V
  by (insert Suc.prems(2) wf-dia-inv, unfold G-def, blast)+
  note lin = linearityD[OF this(1)]

  have acy: ⋀e. e ∈ set E ⟹ fst3 e |∩| thd3 e = {||}
  apply (fold fset-cong, insert ‹acyclicity E›)
  apply (unfold acyclicity-def acyclic-def, auto)
  done

  note lins = lins2D[OF π-in]

  have e-in-E: e ∈ set E
  apply (subgoal-tac set π = (fset G^V − fset S) <+> set G^E)
  apply (unfold π-def x-def G-def edges.simps, auto)[1]
  apply (simp add: lins(2))
  done

  have vs-in-S: vs |⊆| 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 ∈ fset V)
  prefer 2
  apply (auto)
  proof −
    fix v
    assume a: v ∈ fset (fst3 e)
    assume v ∉ fset S and v ∈ fset V
    hence Inl v ∈ set π
    by (metis (lifting) DiffI G-def InlI lins(2) vertices.simps)
    then obtain i where
      i < length π 0 < length π π!i = Inl v  π!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* |∩| (*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* $V'\ \Lambda\ E'$

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* − *S* |=> *Top*] $++_f$ [*S* − *vs* |=> *Bot*]

have *next-initial-ps2*: *initial-ps2* $S'\ G'$
  = *initial-ps2 S G* ⊖ *vs* $++_f$ [*ws* |=> *Top*]
using *next-initial-ps2-edge*[*OF G-def - - - e-in-E - Suc.prems*(*1*)
  Suc.prems(*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* Π' = [*terminals* G' |=> *Bot*]
proof (*intro Suc.hyps*[*of* S'])
  show S' |⊆| *initials* G'
  apply (*insert Suc.prems*(*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.prems*(*2*) **have** *wf-dia* (*Graph V* Λ *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.prems*(*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 Π′ ∈ ps-chains2 S′ G′
      apply (unfold Π-def Π′-def ps-chains2-def)
      apply (intro imageI)
      apply (unfold S′-def G′-def V′-def E′-def)
      apply (intro next-lins2-edge)
      apply (metis π-def G-def x-def π-in)
      by (simp only: vs-in-S e-def)+
    next
      have vs |⊆| V by (metis (lifting) ‹S |⊆| V› order-trans vs-in-S)
      have distinct E using ‹linearity E› linearity-def by auto
      show fcard G′^V + length G′^E = k + fcard S′
      apply (insert Suc.prems(4))
      apply (unfold G-def G′-def vertices.simps edges.simps)
      apply (unfold V′-def E′-def S′-def)
      apply (unfold fcard-funion-fsubset[OF ‹vs |⊆| V›])
      apply (unfold fcard-funion-fsubset[OF ‹vs |⊆| S›])
      apply (fold distinct-remove1-removeAll[OF ‹distinct E›])
      apply (unfold length-remove1)
      apply (simp add: e-in-E)
      apply (drule arg-cong[of - - λx. x − fcard vs − 1])
      apply (subst (asm) add-diff-assoc2[symmetric])
      apply (simp add: fcard-mono[OF ‹vs |⊆| V›])
        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 ‹vs |⊆| S›])
      done
    qed
    hence
      wf-Π′: wf-ps-chain Π′ and
      post-Π′: post Π′ = [ terminals G′ |=> 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 |=> Bot ] |∩|
      fmdom([ ffilter (λv. ∀ e∈set E. v |∉| thd3 e) V − S |=> Top ]
      ++_f [ S − fst3 e |=> 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* Π
using [[*unfold-abs-def* = *false*]]
apply (*unfold* Π-*def* π-*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* Π'-*def*)
apply (*unfold wf-ps-chain-def chain-all.simps conj-commute*)
apply (*intro conjI*)
apply (*fold wf-ps-chain-def*)
apply (*rule wf*-Π')
apply (*intro wf-ps-triple-edgeI exI*[*of* - σ])
apply (*unfold e-def fst3-simp thd3-simp* σ-*def*, *intro conjI*)
apply (*insert Suc.prems*(*1*))
apply (*unfold pre*-Π' *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* |=> *Bot*]
[S − *fst3 e* |=> *Bot*] ($++_f$)])
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 Π ∈ *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*] ⇒ *interface*
where
 ps-to-int G σ ≡
  ⨂ *v* |∈| *fmdom* σ. *case-topbot top-ass bot-ass* (*lookup* σ *v*) (*G*^Λ *v*)

definition
 ps-chain-to-int-chain :: [*diagram*, *ps-chain*] ⇒ *int-chain*
where

ps-chain-to-int-chain $G\ \Pi \equiv$
chainmap (*ps-to-int* G) ((*case-sum* (*Inl* $\circ$ G^Λ) (*Inr* $\circ$ *snd3*))) Π

lemma *ps-chain-to-int-chain-simp*:
ps-chain-to-int-chain (*Graph* $V\ \Lambda\ E$) Π =
chainmap (*ps-to-int* (*Graph* $V\ \Lambda\ E$)) ((*case-sum* (*Inl* $\circ\ \Lambda$) (*Inr* $\circ$ *snd3*))) Π
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 (*asn* P, c, *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 (*asn* P, c, *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 (*asn* P, c, *asn* Q) is provable for each command c that can be extracted from A.

lemma *soundness-graphical-helper*:
assumes *no-var-interference*: $\bigwedge c$. *wr-com* c = {}
shows
(*prov-dia* $G\ P\ Q \longrightarrow$
(P = *top-dia* $G \wedge Q$ = *bot-dia* $G \wedge$
($\forall c$. *coms-dia* $G\ c \longrightarrow$ *prov-triple* (*asn* P, c, *asn* Q))))
$\wedge$ (*prov-com* $C\ P\ Q \longrightarrow$
($\forall c$. *coms-com* $C\ c \longrightarrow$ *prov-triple* (*asn* P, c, *asn* Q)))
$\wedge$ (*prov-ass* $A \longrightarrow$
($\forall c$. *coms-ass* $A\ c \longrightarrow$ *prov-triple* (*asn* (*top-ass* A), c, *asn* (*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 Λ E
    fix π::lin
    fix cs::command list
    assume wf-G: wf-dia (Graph V Λ E)
    assume ⋀v. v ∈ fset V ⟹ ∀c. coms-ass (Λ v) c ⟶
      prov-triple (asn (top-ass (Λ v)), c, asn (bot-ass (Λ v)))
    hence prov-vertex: ⋀v c P Q F. ⟦ coms-ass (Λ v) c; v ∈ fset V;
      P = (top-ass (Λ v) ⊗ F) ; Q = (bot-ass (Λ v) ⊗ F) ⟧
      ⟹ prov-triple (asn P, c, asn Q)
    by (auto simp add: prov-triple.frame no-var-interference)
    assume ⋀e. e ∈ set E ⟹ ∀c. coms-com (snd3 e) c ⟶ prov-triple
      (asn (⨂v|∈|fst3 e. bot-ass (Λ v)),c,asn (⨂v|∈|thd3 e. top-ass (Λ v)))
    hence prov-edge: ⋀e c P Q F. ⟦ e ∈ set E ; coms-com (snd3 e) c ;
      P = ((⨂v|∈|fst3 e. bot-ass (Λ v)) ⊗ F) ;
      Q = ((⨂v|∈|thd3 e. top-ass (Λ v)) ⊗ F) ⟧
      ⟹ prov-triple (asn P, c, asn Q)
    by (auto simp add: prov-triple.frame no-var-interference)
    assume len-cs: length cs = length π
    assume ∀i<length π.
      case-sum (coms-ass ∘ Λ) (coms-com ∘ snd3) (π ! i) (cs ! i)
    hence π-cs: ⋀i. i < length π ⟹
      case-sum (coms-ass ∘ Λ) (coms-com ∘ snd3) (π ! i) (cs ! i) by auto
    assume G-def: G = Graph V Λ E
    assume c-def: c = foldr (;;) cs Skip
    assume π-lin: π ∈ lins (Graph V Λ E)
```

note *lins* = *linsD*[*OF* π-*lin*]

define Π **where** Π = *mk-ps-chain* $\{\!|$ *initial-ps G* $|\!\}$ π

have $\Pi \in$ *ps-chains G* **by** (*simp add*: π-*lin* Π-*def ps-chains-def G-def*)
hence *1*: *post* Π = [*terminals G* |=> *Bot*]
  and *2*: *chain-all wf-ps-triple* Π
by (*insert wf-chains G-def wf-G*, *auto simp add*: *wf-ps-chain-def*)

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

    have *snd3* (*nthtriple* Π *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*-Π*i wf-ps-triple-def* **obtain** σ **where**
v-notin-σ: *v* |∉| *fmdom* σ **and**
fst-Π*i*: *fst3* (*nthtriple* Π *i*) = [{|*v*|} |=> *Top*] $++_f$ σ **and**
thd-Π*i*: *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*-Π*i thd*-Π*i*)
apply (*unfold ps-to-int-def*)
apply (*unfold fmdom-add fdom-make-fmap*)
apply (*unfold finsert-is-funion*[*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*-Π*i 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-Π*i*: *fst3* (*nthtriple* Π *i*) = [*fst3 e* |=> *Bot*] $++_f$ σ **and**
thd-Π*i*: *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 $e \in$ *set E*
apply (*subgoal-tac Inr e* $\in$ *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*)) = (($\bigotimes$ *v*|$\in$|*fst3 e. bot-ass* (Λ *v*)) $\otimes$ *?F*)
unfolding *fst-*Π*i 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* $\forall$ *v* $\in$ *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* |$\notin$| *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*)) = (($\bigotimes$ *v*|$\in$|*thd3 e. top-ass* (Λ *v*)) $\otimes$ *?F*)
unfolding *thd-*Π*i 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* $\forall$ *v* $\in$ *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* $\forall$ *v* $\in$ *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 $\bigwedge c.\ wr\text{-}com\ c = \{\}$
  assumes *prov-dia G P Q*
  shows $\forall\, c.\ coms\text{-}dia\ G\ c \longrightarrow prov\text{-}triple\ (asn\ P,\ c,\ asn\ Q)$
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.