Ribbon Proofs for Separation Logic
(Isabelle Formalisation)

John Wickerson
September 13, 2023

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 . . . . . . . . . . . . . . . . . . 5
4 Proof chains 6
4.1 Projections . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6
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 . . . . . . . . . . . . . . . . 8
4.6 Extending a chain on its right-hand side . . . . . . . . . . . . 8


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 |∩| B = {||} ←→ (∀ x. x |∈| A → x |∉| B)
 ⟨proof⟩
 unbundle lifting-syntax
 unbundle fmap.lifting

type-notation fmap (infix → f 9)

2.1 Difference

definition
  map-diff :: ('k → 'v) ⇒ 'k fset ⇒ ('k → 'v)
where
  map-diff f ks = restrict-map f (− fset ks)

lift-definition
  fmap-diff :: ('k → f 'v) ⇒ 'k fset ⇒ ('k → f 'v) (infix ⊗ 110)
is map-diff
 ⟨proof⟩

2.2 Comprehension

definition
  make-map :: 'k fset ⇒ 'v ⇒ ('k → 'v)
where
  make-map ks v ≡ λk. if k ∈ fset ks then Some v else None
lemma make-map-transfer[transfer-rule]: (rel-fset (=) ===> A ===> rel-map A) make-map make-map
⟨proof⟩

lemma dom-make-map:
  dom (make-map ks v) = fset ks
⟨proof⟩

lift-definition
  make-fmap :: ′k fset ⇒ ′v ⇒ ('k => ′v) [[ - => - ])
is make-map parametric make-map-transfer
⟨proof⟩

lemma make-fmap-empty[simp]: [ {} => f ] = fmempty
⟨proof⟩

2.3 Domain

lemma fmap-add-commute:
  assumes fmdom A [\cap] fmdom B = {}
  shows A ++ f B = B ++ f A
⟨proof⟩ including fset.lifting
⟨proof⟩

lemma make-fmap-union:
  [ xs => v ] ++ f [ ys => v ] = [ xs \cup ys => v ]
⟨proof⟩

lemma fdom-make-fmap: fmdom [ ks => v ] = ks
⟨proof⟩

2.4 Lookup

lift-definition
  lookup :: ('k => f 'v) ⇒ 'k ⇒ 'v
is (⊙) the ⟨proof⟩

lemma lookup-make-fmap:
  assumes k ∈ fset ks
  shows lookup [ ks => v ] k = v
⟨proof⟩

lemma lookup-make-fmap1:
  lookup [ {}|k| ] => v ] k = v
⟨proof⟩

lemma lookup-union1:
  assumes k ∈ fdom ys
  shows lookup (xs ++ f ys) k = lookup ys k
proof including fset.lifting

lemma lookup-union2:
  assumes k \notin fdom ys
  shows lookup (xs ++ f ys) k = lookup xs k
proof including fset.lifting
proof

lemma lookup-union3:
  assumes k \notin fdom xs
  shows lookup (xs ++ f ys) k = lookup ys k
proof including fset.lifting
proof

end

3 General purpose definitions and lemmas

theory JHelper imports Main begin

lemma Collect-iff:
a \in \{ x . P x \} \equiv P a
proof

lemma diff-diff-eq:
  assumes C \subseteq B
  shows (A - C) - (B - C) = A - B
proof

lemma nth-in-set:
  [ i < length xs ; xs ! i = x ] \Rightarrow x \in set xs
proof

lemma disjI [intro]:
  assumes \neg P \Rightarrow Q
  shows P \lor Q
proof

lemma empty-eq-Plus-conv:
  (\{\} = A <+> B) = (A = \{\} \land B = \{\})
proof

3.1 Projection functions on triples

definition fst3 :: 'a \times 'b \times 'c \Rightarrow 'a
where fst3 \equiv fst
\textbf{definition} \texttt{snd3} :: \texttt{\textquotesingle a \times \textquotesingle b \times \textquotesingle c} \Rightarrow \texttt{\textquotesingle b} \\
\textbf{where} \texttt{snd3} \equiv \texttt{fst} \circ \texttt{snd}

\textbf{definition} \texttt{thd3} :: \texttt{\textquotesingle a \times \textquotesingle b \times \textquotesingle c} \Rightarrow \texttt{\textquotesingle c} \\
\textbf{where} \texttt{thd3} \equiv \texttt{snd} \circ \texttt{snd}

\textbf{lemma} \texttt{fst3-simp}: \\
\forall a \ b \ c. \texttt{fst3} (a,b,c) = a \\
\langle \text{proof} \rangle

\textbf{lemma} \texttt{snd3-simp}: \\
\forall a \ b \ c. \texttt{snd3} (a,b,c) = b \\
\langle \text{proof} \rangle

\textbf{lemma} \texttt{thd3-simp}: \\
\forall a \ b \ c. \texttt{thd3} (a,b,c) = c \\
\langle \text{proof} \rangle

\textbf{lemma} \texttt{tripleI}: \\
\texttt{fixes} T U \\
\texttt{assumes} \texttt{fst3} T = \texttt{fst3} U \\
\texttt{and} \texttt{snd3} T = \texttt{snd3} U \\
\texttt{and} \texttt{thd3} T = \texttt{thd3} U \\
\texttt{shows} T = U \\
\langle \text{proof} \rangle

\textbf{end}

\section{4 \ Proof chains}

\textbf{theory} \texttt{Proofchain imports} \\
\texttt{JHelper} \\
\texttt{begin}

An \texttt{\textquotesingle a \textquotesingle c} chain is a sequence of alternating \texttt{\textquotesingle a\textquotesingle}s and \texttt{\textquotesingle c\textquotesingle}s, beginning and ending with an \texttt{\textquotesingle a}. Usually \texttt{\textquotesingle a} represents some sort of assertion, and \texttt{\textquotesingle c} represents some sort of command. Proof chains are useful for stating the \texttt{SMain} proof rule, and for conducting the proof of soundness.

\textbf{datatype} \texttt{(\textquotesingle a \textquotesingle c) chain} = \\
\texttt{cNil \textquotesingle a} \\
\texttt{cCons \textquotesingle a \textquotesingle c (\textquotesingle a \textquotesingle c) chain} \\
\texttt{\texttt{\textbackslash n e w \textbackslash C o n s \textquotesingle a \textquotesingle c (\textquotesingle a \textquotesingle c) chain}} \\
\texttt{\{ 0 , 0 , 0 , \ldots , 0 \} 60} \\

For example, \texttt{\textbackslash c C o n s \textbackslash a \textbackslash c (\textbackslash a \textbackslash c) chain} \texttt{\textbackslash n e w} \texttt{C o n s \textbackslash a \textbackslash c (\textbackslash a \textbackslash c) chain} \texttt{\{ 0 , 0 , 0 \} \ldots \{ 0 , 0 , 0 \} 60}.

\subsection{4.1 \ Projections}

Project first assertion.
fun
  pre :: ('a,'c) chain ⇒ 'a
where
  pre² P = P
  | pre (↑ P · · · ) = P

Project final assertion.

fun
  post :: ('a,'c) chain ⇒ 'a
where
  post² P = P
  | post (↑ · · · II) = post II

Project list of commands.

fun
  comlist :: ('a,'c) chain ⇒ 'c list
where
  comlist² [] = []
  | comlist (↑ x · Π) = x # (comlist Π)

4.2 Chain length

fun
  chainlen :: ('a,'c) chain ⇒ nat
where
  chainlen² [] = 0
  | chainlen (↑ · · · II) = 1 + (chainlen II)

lemma len-comlist-chainlen:
  length (comlist II) = chainlen II
⟨proof⟩

4.3 Extracting triples from chains

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

fun
  nthtriple :: ('a,'c) chain ⇒ nat ⇒ ('a * 'c * 'a)
where
  nthtriple² P x Π 0 = (P, x, pre Π)
  | nthtriple² P x Π (Suc n) = nthtriple Π 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 < chainlen II ⇒ snd³ (nthtriple Π i) = (comlist Π)!i
⟨proof⟩
4.4 Evaluating a predicate on each triple of a chain

chain-all \varphi\ holds of \Pi iff \varphi\ holds for each of \Pi's individual triples.

fun chain-all :: (('a × 'c × 'a) \Rightarrow bool) \Rightarrow ('a,'c) chain \Rightarrow bool
where
  chain-all \varphi \{σ\} = True
| chain-all \varphi (\{σ\} · x · Π) = (\varphi (σ,pre Π) \land chain-all \varphi Π)

lemma chain-all-mono [mono]:
  \(x \leq y \Rightarrow chain-all x \leq chain-all y\)
⟨proof⟩

lemma chain-all-nthtriple:
  (chain-all \varphi Π) = (\forall i < chainlen Π. \varphi (nthtriple Π i))
⟨proof⟩

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 ⇒ 'b) ⇒ ('c ⇒ 'd) ⇒ ('a,'c) chain ⇒ ('b,'d) chain
where
  chainmap f g \{P\} = \{f P\}
| chainmap f g (\{P\} · x · Π) = \{f P\} · g x · chainmap f g Π

Mapping over a chain preserves its length.

lemma chainmap-preserves-length:
  chainlen (chainmap f g Π) = chainlen Π
⟨proof⟩

lemma pre-chainmap:
  pre (chainmap f g Π) = f (pre Π)
⟨proof⟩

lemma post-chainmap:
  post (chainmap f g Π) = f (post Π)
⟨proof⟩

lemma nthtriple-chainmap:
  assumes i < chainlen Π
  shows nthtriple (chainmap f g Π) i
    = (\lambda t. (f (fst3 t), g (snd3 t), f (thd3 t))) (nthtriple Π i)
⟨proof⟩

4.6 Extending a chain on its right-hand side

fun
cSnoc :: ('a,'c) chain ⇒ 'c ⇒ 'a ⇒ ('a,'c) chain

where

cSnoc (σ · σ) y τ = {σ} · y · {τ}
cSnoc (σ · x · Π) y τ = {σ} · x · (cSnoc Π y τ)

lemma len-snoc:
  fixes Π x P
  shows chainlen (cSnoc Π x P) = 1 + (chainlen Π)
⟨proof⟩

lemma pre-snoc:
  pre (cSnoc Π x P) = pre Π
⟨proof⟩

lemma post-snoc:
  post (cSnoc Π x P) = P
⟨proof⟩

lemma comlist-snoc:
  comlist (cSnoc Π x P) = comlist Π @ [x]
⟨proof⟩

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 ⋆ q = q ⋆ p and
    star-assoc: (p ⋆ q) ⋆ r = p ⋆ (q ⋆ r) and
    star-emp: p ⋆ Emp = p and
    emp-star: Emp ⋆ p = p

lemma star-rot:
    q ⋆ p ⋆ r = p ⋆ q ⋆ r
⟨proof⟩

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 ⋆ 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 \cup rd-com c2

Extracting the set of program variables written by a command.

axiomatization

wr-com :: command \Rightarrow string set

where wr-com-choose: wr-com (Choose c1 c2) = wr-com c1 \cup 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 \cup 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 \times command \times assertion \Rightarrow bool

where

events: prov-triple (p, c, q) \Rightarrow prov-triple (\{ x \mid p, c, q \})
| choose: \{ prov-triple (p, c1, q); prov-triple (p, c2, q) \} \Rightarrow prov-triple (p, \{ x \mid p, c1, c2, q \})
| loop: prov-triple (p, c, p) \Rightarrow prov-triple (p, Loop c, p)
| frame: \{ prov-triple (p, c, q); wr-com(c) \cap rd-ass(r) = \{\} \} \Rightarrow prov-triple (p, r \times c, q \times r)
| skip: prov-triple (p, Skip, p)
| seq: \{ prov-triple (p, c1, q); prov-triple (q, c2, r) \} \Rightarrow 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, \{ x \mid p, c1, c2, q \})

lemma loop-lemma:
assumes prov-triple (p1, c, q1) and p = p1 and p1 = p2 and q = q1 and q1 = q2
shows prov-triple (p, Loop c, q)

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)

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 asso-
ciativity, 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 (\varepsilon_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 \varepsilon_c the left and right unit of (\otimes_c).

inductive equiv-int :: conc-interface \Rightarrow conc-interface \Rightarrow bool (infix \righteq_c 45)
where
  refl: P \righteq P
  sym: P \righteq Q \Longrightarrow Q \righteq P
  trans: [P \righteq Q; Q \righteq R] \Longrightarrow P \righteq R
  cong-hcomp1: P \righteq Q \Longrightarrow P' \otimes_c P \righteq P' \otimes_c Q
  cong-hcomp2: P \righteq Q \Longrightarrow P \otimes_c P' \righteq Q \otimes_c P'
  cong-exists: P \righteq Q \Longrightarrow \text{Exists-int-conc x P} \righteq \text{Exists-int-conc x Q}
  hcomp-conc-associ: P \otimes_c (Q \otimes_c R) \righteq (P \otimes_c Q) \otimes_c R
  hcomp-conc-comm: P \otimes_c Q \righteq Q \otimes_c P
  hcomp-conc-unit1: \varepsilon_c \otimes_c P \righteq P
  hcomp-conc-unit2: P \otimes_c \varepsilon_c \righteq P

lemma equiv-int-cong-hcomp:
  \[ P \righteq Q ; P' \righteq Q' \] \Longrightarrow P \otimes_c P' \righteq Q \otimes_c Q'
  (proof)

quotient-type interface = conc-interface / equiv-int
  (proof)
lift-definition
  \textit{Ribbon} :: \textit{assertion} \Rightarrow \textit{interface}
is \textit{Ribbon-conc} \langle \textit{proof} \rangle

lift-definition
  \textit{Emp-int} :: \textit{interface} (\varepsilon)
is \varepsilon_c \langle \textit{proof} \rangle

lift-definition
  \textit{Exists-int} :: \textit{string} \Rightarrow \textit{interface} \Rightarrow \textit{interface}
is \textit{Exists-int-conc}
\langle \textit{proof} \rangle

lift-definition
  \textit{HComp-int} :: \textit{interface} \Rightarrow \textit{interface} \Rightarrow \textit{interface} (\textit{infix} \otimes 50)
is \textit{HComp-int-conc} \langle \textit{proof} \rangle

lemma \textit{hcomp-comm}:
  \((P \otimes Q) = (Q \otimes P)\)
\langle \textit{proof} \rangle

lemma \textit{hcomp-assoc}:
  \((P \otimes (Q \otimes R)) = ((P \otimes Q) \otimes R)\)
\langle \textit{proof} \rangle

lemma \textit{emp-hcomp}:
  \(\varepsilon \otimes P = P\)
\langle \textit{proof} \rangle

lemma \textit{hcomp-emp}:
  \(P \otimes \varepsilon = P\)
\langle \textit{proof} \rangle

lemma \textit{comp-fun-commute-hcomp}:
  \textit{comp-fun-commute} (\otimes)
\langle \textit{proof} \rangle

\subsection*{6.2 An iterated horizontal-composition operator}
definition \textit{iter-hcomp} :: ('a fset) \Rightarrow ('a \Rightarrow \textit{interface}) \Rightarrow \textit{interface}
where
  \textit{iter-hcomp} X f \equiv \text{fold} ((\otimes) \circ f) \varepsilon X

syntax \textit{iter-hcomp-syntax} ::
  ('a \Rightarrow ('a fset)) \Rightarrow ('a \Rightarrow \textit{interface}) \Rightarrow \textit{interface}
  \((\otimes x|\in|M. e \mapsto CONST \textit{iter-hcomp} M (\lambda x. e))\)
term $\bigotimes P \in \mathcal{P}_s. f P$ — this is eta-expanded, so prints in expanded form

term $\bigotimes P \in \mathcal{P}_s. f$ — this isn’t eta-expanded, so prints as written

**Lemma iter-hcomp-cong:**
- **Assumes** $\forall v \in \text{fset } vs. \; \phi = \phi'$
- **Shows** $(\bigotimes v \in vs. \; \phi) = (\bigotimes v \in vs. \; \phi')$

**Lemma iter-hcomp-empty:**
- **Shows** $(\bigotimes x \in \{\} . \; p x) = \varepsilon$

**Lemma iter-hcomp-insert:**
- **Assumes** $v \not\in \mathcal{W}$
- **Shows** $(\bigotimes x \in \mathcal{F}_\text{insert } v \; \mathcal{W} . \; p x) = (p \; v \otimes (\bigotimes x \in \mathcal{W} . \; p x))$

**Lemma iter-hcomp-union:**
- **Assumes** $\mathcal{V} \cap \mathcal{W} = \{\}$
- **Shows** $(\bigotimes x \in \mathcal{V} \cup \mathcal{W} . \; p x) = ((\bigotimes x \in \mathcal{V} . \; p x) \otimes (\bigotimes x \in \mathcal{W} . \; p x))$

### 6.3 Semantics of Interfaces

The semantics of an interface is an assertion.

**Fun**

\[
\text{conc-asn} :: \text{conc-interface} \Rightarrow \text{assertion}
\]

**Where**

\[
\text{conc-asn} (\text{Ribbon-conc } p) = p \\
\text{conc-asn} (P \otimes_c Q) = (\text{conc-asn } P) \star (\text{conc-asn } Q) \\
\text{conc-asn } (\varepsilon_c) = \text{Emp} \\
\text{conc-asn} (\text{Exists-int-conc } x P) = \text{Exists } x \; (\text{conc-asn } P)
\]

**Lift-definition**

\[
\text{asn} :: \text{interface} \Rightarrow \text{assertion}
\]

**Is** \text{conc-asn}

**Lemma** \text{asn-simps} [simp]:

\[
\text{asn} (\text{Ribbon } p) = p \\
\text{asn} (P \otimes Q) = (\text{asn } P) \star (\text{asn } Q) \\
\text{asn } \varepsilon = \text{Emp} \\
\text{asn} (\text{Exists-int } x P) = \text{Exists } x \; (\text{asn } P)
\]

### 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 ⊗_e 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
⟨proof⟩

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
⟨proof⟩

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

lemma seq-fold:
∀Π. length cs = chainlen Π ; p1 = asn (pre Π) ; p2 = asn (post Π) ;
∀i. i < chainlen Π ⇒ prov-triple
(asn (fst3 (nthtriple Π i)), cs ! i, asn (thd3 (nthtriple Π i))) {}
⇒ prov-triple (p1, foldr (;;) cs Skip, p2)
⟨proof⟩

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 o 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) = (∪ 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 (∪ ϱ ∈ set ϱs. wr-cell (fst ϱ))
    = wr-com (foldr (;;) (map (λ(γ,F). com-cell γ) ϱs) Skip)
  and wr-cell (fst ρ) = wr-com (com-cell (fst ρ))
⟨proof⟩

7.2 Proof rules for stratified diagrams
inductive
  prov-sdia :: [sdiagram, interface, interface] ⇒ bool and
  prov-row :: [row, interface, interface] ⇒ bool and
  prov-cell :: [cell, interface, interface] ⇒ bool
where
SRibbon: prov-cell (Filler P) P P
7.3 Soundness

**Lemma soundness-strat-helper:**

\[(prov-sdia D P Q) \rightarrow prov-triple (asn P, com-sdia D, asn Q)) \land
\[(prov-row \varrho P Q \rightarrow prov-triple (asn P, com-cell (fst \varrho), asn Q)) \land
\[(prov-cell \gamma P Q \rightarrow prov-triple (asn P, com-cell \gamma, asn Q))\]

⟨proof⟩

**Corollary soundness-strat:**

assumes prov-sdia D P Q

shows prov-triple (asn P, com-sdia D, asn Q)

⟨proof⟩

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

typedec 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¬V) = (a test)¬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 |∩| fset3 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 |∩| fset3 f = {||} ∧
    thd3 e |∩| thd3 f = {||}
⟨proof⟩

lemma linearityD2:
linearity E → (∀ e f. e ∈ set E ∧ f ∈ set E ∧ e ≠ f →
    fst3 e |∩| fset3 f = {||} ∧
    thd3 e |∩| thd3 f = {||})
⟨proof⟩
inductive

\[ \text{wf-ass :: assertion-gadget} \Rightarrow \text{bool and} \]
\[ \text{wf-com :: command-gadget} \Rightarrow \text{bool and} \]
\[ \text{wf-dia :: diagram} \Rightarrow \text{bool} \]

where

\[ \text{wf-rib: wf-ass (Rib p)} \]
\[ \text{wf-exists: wf-dia G} \Rightarrow \text{wf-ass (Exists-dia \( x \) G)} \]
\[ \text{wf-com: wf-com (Com c)} \]
\[ \text{wf-choice: \[ \text{wf-dia G ; wf-dia H \] } \Rightarrow \text{wf-com (Choose-dia G H)} \]
\[ \text{wf-loop: wf-dia G} \Rightarrow \text{wf-com (Loop-dia G)} \]
\[ \text{wf-dia: \[ \forall e \in \text{set } E. \text{wf-com (snd3 e) ; } \forall v \in \text{fset } V. \text{wf-ass (\( \Lambda \) v)} ; \text{acyclicity } E ; \text{linearity } E ; \forall e \in \text{set } E. \text{fst3 e} | \cup | \text{thd3 e} | \subseteq | V \] } \Rightarrow \]

\[ \text{wf-dia (Graph } V \ \Lambda \ E \) \]

inductive-cases \( \text{wf-dia-inv'}: \text{wf-dia (Graph } V \ \Lambda \ E \) \]

lemma \( \text{wf-dia-inv}: \)
\[ \text{assumes \text{wf-dia (Graph } V \ \Lambda \ E \) \]
\[ \text{shows } \forall v \in \text{fset } V. \text{wf-ass (\( \Lambda \) v)} \]
\[ \text{and } \forall e \in \text{set } E. \text{wf-com (snd3 e)} \]
\[ \text{and } \text{acyclicity } E \]
\[ \text{and } \text{linearity } E \]
\[ \text{and } \forall e \in \text{set } E. \text{fst3 e} | \cup | \text{thd3 e} | \subseteq | V \]
\( \langle \text{proof} \rangle \)

8.3 Initial and terminal nodes

definition
\[ \text{initials :: diagram} \Rightarrow \text{node fset} \]

where
\[ \text{initials } G = \text{ffilter (\( \lambda v. (\forall e \in \text{set } G^E. v | \notin | \text{thd3 e}) \)) } G^V \]

definition
\[ \text{terminals :: diagram} \Rightarrow \text{node fset} \]

where
\[ \text{terminals } G = \text{ffilter (\( \lambda v. (\forall e \in \text{set } G^E. v | \notin | \text{fst3 e}) \)) } G^V \]

lemma \( \text{no-edges-imp-all-nodes-initial}: \)
\[ \text{initials (Graph } V \ \Lambda \ []\} = V \]
\( \langle \text{proof} \rangle \)

lemma \( \text{no-edges-imp-all-nodes-terminal}: \)
\[ \text{terminals (Graph } V \ \Lambda \ []\} = V \]
\( \langle \text{proof} \rangle \)

lemma \( \text{initials-in-vertices}: \)
\[ \text{initials } G | \subseteq | G^{-V} \]
\( \langle \text{proof} \rangle \)
lemma terminals-in-vertices:
  terminals G ⊆ G^V
(proof)

8.4 Top and bottom interfaces

primrec
  top-ass :: assertion-gadget ⇒ interface and
top-dia :: diagram ⇒ interface
where
  top-dia (Graph V Λ E) = (∏ v |∈| 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)
  \And 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** \( \text{lin} = (\text{node} + \text{edge}) \text{ list} \)

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** \( \text{lins} :: \text{diagram} \Rightarrow \text{lin set} \) where

\[
\text{lins} \; G \equiv \{ \pi :: \text{lin}. \; (\text{distinct} \; \pi) \land (\text{set} \; \pi = (\text{set} \; G^V) <+> (\text{set} \; G^E)) \land (\forall \; i \; j \; v \; e. \; i < \text{length} \; \pi \land j < \text{length} \; \pi \land \pi!i = \text{Inl} \; v \land \pi!j = \text{Inr} \; e \land v \; |\in| \; \text{fst3} \; e \rightarrow i < j) \land (\forall \; j \; k \; w \; e. \; j < \text{length} \; \pi \land k < \text{length} \; \pi \land \pi!j = \text{Inr} \; e \land \pi!k = \text{Inl} \; w \land w \; |\in| \; \text{thd3} \; e \rightarrow j < k) \} \]

**lemma** \( \text{linsD} \):

**assumes** \( \pi \in \text{lins} \; G \)**
**shows** (distinct \( \pi \)) and (set \( \pi = (\text{set} \; G^V) <+> (\text{set} \; G^E) \)) and (\( \forall \; i \; j \; v \; e. \; i < \text{length} \; \pi \land j < \text{length} \; \pi \land \pi!i = \text{Inl} \; v \land \pi!j = \text{Inr} \; e \land v \; |\in| \; \text{fst3} \; e \rightarrow i < j) \land (\forall \; j \; k \; w \; e. \; j < \text{length} \; \pi \land k < \text{length} \; \pi \land \pi!j = \text{Inr} \; e \land \pi!k = \text{Inl} \; w \land w \; |\in| \; \text{thd3} \; e \rightarrow j < k) \)

**⟨proof⟩**

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

**lemma** \( \text{coms-mono-helper} \):

\[
(\forall \; i < \text{length} \; \pi. \; \text{case-sum} \; (\text{coms-ass} \circ \Lambda) \; (\text{coms-com} \circ \text{snd3}) \; (\pi!i) \; (\text{cs}!i))
= (\forall \; i < \text{length} \; \pi \land (\exists \; v. \; (\pi!i) = \text{Inl} \; v) \rightarrow \text{coms-ass} \; (\Lambda \; (\text{projl} \; (\pi!i))) \; (\text{cs}!i)) \land (\forall \; i < \text{length} \; \pi \land (\exists \; e. \; (\pi!i) = \text{Inr} \; e) \rightarrow \text{coms-com} \; (\text{snd3} \; (\text{projr} \; (\pi!i))) \; (\text{cs}!i)) \]

**⟨proof⟩**

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

**inductive** \( \text{coms-dia} :: [\text{diagram}, \text{command}] \Rightarrow \text{bool and} \)
coms-ass :: [assertion-gadget, command] ⇒ bool and
coms-com :: [command-gadget, command] ⇒ bool

where
coms-skip: coms-ass (Rib p) Skip
coms-exists: coms-dia G c ⇒ coms-ass (Exists-dia x G) c
coms-basic: coms-com (Com c) c
coms-choice: [ coms-dia G c; coms-dia H d ] ⇒ coms-com (Choose-dia G H) (Choose c d)
coms-loop: coms-dia G c ⇒ coms-com (Loop-dia G) (Loop c)
coms-main: [ π ∈ lins (Graph V Λ E); length cs = length π; ∀ i<length π. case-sum (coms-ass ◦ Λ) (coms-com ◦ snd3) (π!i) (cs!i) ] ⇒ 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.

```haskell
datatype topbot = Top | Bot

type-synonym proofstate = node ↦ 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.

```haskell
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.

```haskell
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.

```haskell
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⟩

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
  ⟨proof⟩

Distributing `mk-ps-chain` over snoc.

lemma mk-ps-chain-snoc:
\[ \text{mk-ps-chain } \Pi (\pi @ [x]) = \text{cSnoc (mk-ps-chain } \Pi \pi) x \ (\text{next-ps (post (mk-ps-chain } \Pi \pi)) x) \]

Distributing \text{mk-ps-chain} over \text{cCons}.

**lemma** \text{mk-ps-chain-ccons}:

fixes \Pi \pi

shows \text{mk-ps-chain} (\langle \sigma \rangle \cdot x \cdot \Pi) \pi = \langle \sigma \rangle \cdot x \cdot \text{mk-ps-chain } \Pi \pi

**lemma** \text{pre-mk-ps-chain}:

fixes \Pi \pi

shows \text{pre (mk-ps-chain } \Pi \pi) = \text{pre } \Pi

A chain which is obtained from the list \pi, has \pi 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** \text{comlist-mk-ps-chain}:

\text{comlist (mk-ps-chain } \Pi \pi) = \text{comlist } \Pi @ \pi

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 \text{lins G} function. This was originally defined in \text{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** \text{lins2 :: [node fset, diagram] ⇒ lin set}

\[
\text{lins2 } S \ G \equiv \{ \pi :: \text{lin} . \ (\text{distinct } \pi) \land (\text{set } \pi = (\text{fset } G^V - \text{fset } S) \land \text{set } G^E) \land (\forall i j v e. \ i < \text{length } \pi \land j < \text{length } \pi \land \pi!i = \text{Inl } v \land \pi!j = \text{Inr } e \land v \mid e \mid \text{fst3 } e \rightarrow i < j) \land (\forall j k w e. \ j < \text{length } \pi \land k < \text{length } \pi \land \pi!j = \text{Inr } e \land \pi!k = \text{Inl } w \land w \mid e \mid \text{thd3 } e \rightarrow j < k) \} \]

24
lemma lins2D:
  assumes π ∈ lins2 S G
  shows distinct π
  and set π = (fset G^V − fset S) <+> set G^E
  and \( \lambda i j v e . [ i < \text{length} \pi ; j < \text{length} \pi ; \pi!i = \text{Inl} v ; \pi!j = \text{Inr} e ; v \in| \text{fst3} e ] \implies i < j \)
  and \( \lambda i k w e . [ j < \text{length} \pi ; k < \text{length} \pi ; \pi!j = \text{Inr} e ; \pi!k = \text{Inl} w ; w \in| \text{thd3} e ] \implies j < k \)
⟨proof⟩

lemma lins2I:
  assumes distinct π
  and set π = (fset G^V − fset S) <+> set G^E
  and \( \lambda i j v e . [ i < \text{length} \pi ; j < \text{length} \pi ; \pi!i = \text{Inl} v ; \pi!j = \text{Inr} e ; v \in| \text{fst3} e ] \implies i < j \)
  and \( \lambda j k w e . [ j < \text{length} \pi ; k < \text{length} \pi ; \pi!j = \text{Inr} e ; \pi!k = \text{Inl} w ; w \in| \text{thd3} e ] \implies j < k \)
  shows π ∈ lins2 S G
⟨proof⟩

When S is empty, the two definitions coincide.

lemma lins-is-lins2-with-empty-S:
  lins G = lins2 {} G
⟨proof⟩

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

definition initial-ps :: diagram ⇒ proofstate
where
  initial-ps G ≡ [ 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] ⇒ proofstate
where
  initial-ps2 S G ≡ [ 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 {}
⟨proof⟩

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

definition ps-chains :: diagram ⇒ ps-chain set
where
\[ ps\text{-chains} \ G \equiv \text{mk-ps-chain} \ (c\text{Nil} \ (\text{initial-ps} \ G)) \cdot \text{lins} \ G \]

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

definition
\[ ps\text{-chains2} :: [\text{node \ fset}, \text{diagram}] \Rightarrow \text{ps-chain \ set} \]
where
\[ ps\text{-chains2} \ S \ G \equiv \text{mk-ps-chain} \ (c\text{Nil} \ (\text{initial-ps2} \ S \ G)) \cdot \text{lins2} \ S \ G \]

When \( S \) is empty, the above two definitions coincide.

lemma ps-chains-is-ps-chains2-with-empty-S:
\[ ps-chains = ps\text{-chains2} \ {||} \]

We now wish to describe proofstates chain that are well-formed. First, let us say that \( f ++ f \text{ disjoint} \ g \) is defined, when \( f \) and \( g \) have disjoint domains, as \( f ++ f \text{ disjoint} \ g \). Then, a well-formed proofstate chain consists of triples of the form \((\sigma ++ f \text{ disjoint} [\{v\}] \Rightarrow \text{Top}], \text{Inl} \ v, \sigma ++ f \text{ disjoint} [\{v\}] \Rightarrow \text{Bot} \), where \( v \) is a node, or of the form \((\sigma ++ f \text{ disjoint} [\{w\}] \Rightarrow \text{Bot}], \text{Inr} \ e, \sigma ++ f \text{ disjoint} [\{w\}] \Rightarrow \text{Top} \), 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
\[ \text{wf-ps-triple} :: \text{proofstate} \times (\text{node} + \text{edge}) \times \text{proofstate} \Rightarrow \text{bool} \]
where
\[ \text{wf-ps-triple} \ T = (\text{case \ snd3 \ T \ of} \]
\[ \quad \text{Inl} \ v \Rightarrow (\exists \sigma. \ v \not\in f\text{dom} \ \sigma \]
\[ \quad \exists \) \]
\[ \quad \text{fst3} \ T = [\{v\}] \Rightarrow \text{Top} ] ++ f \ \sigma \]
\[ \quad \exists \) \]
\[ \quad \text{thd3} \ T = [\{v\}] \Rightarrow \text{Bot} ] ++ f \ \sigma \]
\[ \quad \exists \) \]
\[ \quad \text{Inr} \ e \Rightarrow (\exists \sigma. \ (\text{fst3} \ e \cup \text{thd3} \ e) \cap f\text{dom} \ \sigma = {||}) \]
\[ \quad \exists \) \]
\[ \quad \text{fst3} \ T = [\text{fst3} \ e \Rightarrow \text{Bot} ] ++ f \ \sigma \]
\[ \quad \exists \) \]
\[ \quad \text{thd3} \ T = [\text{thd3} \ e \Rightarrow \text{Top} ] ++ f \ \sigma) \]

lemma \( \text{wf-ps-triple-nodeI} \):
\[ \exists \sigma. \ v \not\in f\text{dom} \ \sigma \]
\[ \sigma 1 = [\{v\}] \Rightarrow \text{Top} ] ++ f \ \sigma \]
\[ \sigma 2 = [\{v\}] \Rightarrow \text{Bot} ] ++ f \ \sigma \]
shows \( \text{wf-ps-triple} \ (\sigma 1, \text{Inl} \ v, \sigma 2) \)

 lemma \( \text{wf-ps-triple-edgeI} \):
\[ \exists \sigma. \ (\text{fst3} \ e \cup \text{thd3} \ e) \cap f\text{dom} \ \sigma = {||}) \]
\[ \exists \) \]
\[ \sigma 1 = [\text{fst3} \ e \Rightarrow \text{Bot} ] ++ f \ \sigma \]
\[ \exists \) \]
\[ \sigma 2 = [\text{thd3} \ e \Rightarrow \text{Top} ] ++ f \ \sigma \]
shows \( \text{wf-ps-triple} \ (\sigma 1, \text{Inr} \ e, \sigma 2) \)
\[ \text{definition}\]
\[ \text{wf-ps-chain :: ps-chain } \Rightarrow \text{bool} \]
\[ \text{where} \]
\[ \text{wf-ps-chain } \equiv \text{chain-all} \text{ wf-ps-triple} \]

\[ \text{lemma next-initial-ps2-vertex:}\]
\[ \text{initial-ps2} \ (\{|v|\} \cup |S|) G \]
\[ = \text{initial-ps2} \ S G \odot \{|v|\} \mathbf{++} \ [ \{|v|\} \mapsto \text{Bot}] \]
\[ \langle \text{proof}\rangle \]

\[ \text{lemma next-initial-ps2-edge:}\]
\[ \text{assumes } G = \text{Graph} \ V \Lambda \ E \text{ and } G' = \text{Graph} \ V' \Lambda \ E' \text{ and } \]
\[ V = V - \text{fst3} \ e \text{ and } E' = \text{removeAll} \ e \ E \text{ and } e \in \text{set} \ E \text{ and } \]
\[ \text{fst3} \ e \subseteq |S| \text{ and } S \subseteq |\text{initials} \ G| \text{ and } \text{wf-dia} \ G \]
\[ \text{shows } \text{initial-ps2} \ (S - \text{fst3} \ e) G' = \]
\[ \text{initial-ps2} \ S G \odot \text{fst3} \ e \mathbf{++} \ [ \ \text{thd3} \ e \mapsto \text{Top}] \]
\[ \langle \text{proof}\rangle \]

\[ \text{lemma next-lins2-vertex:}\]
\[ \text{assumes } \text{Inl} \ v \# \pi \in \text{lins2} \ S G \]
\[ \text{assumes } v \not\in S \]
\[ \text{shows } \pi \in \text{lins2} \ (\{|v|\} \cup |S|) G \]
\[ \langle \text{proof}\rangle \]

\[ \text{lemma next-lins2-edge:}\]
\[ \text{assumes } \text{Inr} \ e \# \pi \in \text{lins2} \ S \ (\text{Graph} \ V \Lambda E) \]
\[ \text{and } vs \subseteq S \]
\[ \text{and } e = (vs,c,ws) \]
\[ \text{shows } \pi \in \text{lins2} \ (S - vs) \ (\text{Graph} \ (V - vs) \Lambda (\text{removeAll} \ e \ E)) \]
\[ \langle \text{proof}\rangle \]

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 \( S \).

\[ \text{lemma wf-chains2:}\]
\[ \text{fixes } k \]
\[ \text{assumes } S \subseteq |\text{initials} \ G| \]
\[ \text{and } \text{wf-dia} \ G \]
\[ \text{and } \Pi \in \text{ps-chains2} \ S G \]
\[ \text{and } \text{fcard} \ G^V + \text{length} \ G^E = k + \text{fcard} \ S \]
\[ \text{shows } \text{wf-ps-chain} \ \Pi \land (\text{post} \ \Pi = [ \text{terminals} \ G |\mapsto \text{Bot}]) \]
\[ \text{proof} \]

**Corollary** `wf-chains`:
- Assumes `wf-dia G`
- Assumes `\Pi \in \text{ps-chains} G`
- Shows `\text{wf-ps-chain} \Pi \land \text{post} \Pi = [ \text{terminals} G \Rightarrow \text{Bot} ]`

\[ \text{proof} \]

### 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**

```plaintext
ps-to-int :: [diagram, proofstate] \Rightarrow interface
where
ps-to-int G \sigma \equiv
\forall v |\in| \text{fmdom} \sigma. \text{case-topbot top-ass bot-ass} (\text{lookup} \sigma v) (G^\Lambda v)
```

**Definition**

```plaintext
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^\Lambda) (\text{Inr} \circ \text{snd3}))) \Pi)
```

**Lemma** `ps-chain-to-int-chain-simp`:

```plaintext
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{snd3}))) \Pi)
```

\[ \text{proof} \]

### 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: $\forall c. \text{wr-com } c = \{\}$

**Shows**

$$(\text{prov-dia } G P Q \rightarrow (P = \text{top-dia } G \land Q = \text{bot-dia } G \land (\forall c. \text{coms-dia } G c \rightarrow \text{prov-triple } (\text{asn } P, c, \text{asn } Q))))$$

$$\land (\text{prov-com } C P Q \rightarrow (\forall c. \text{coms-com } C c \rightarrow \text{prov-triple } (\text{asn } P, c, \text{asn } Q))))$$

$$\land (\text{prov-ass } A \rightarrow (\forall c. \text{coms-ass } A c \rightarrow \text{prov-triple } (\text{asn } (\text{top-ass } A), c, \text{asn } (\text{bot-ass } A))))$$

**Proof**

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** $\forall c. \text{wr-com } c = \{\}$

**Assumes** $\text{prov-dia } G P Q$

**Shows** $\forall c. \text{coms-dia } G c \rightarrow \text{prov-triple } (\text{asn } P, c, \text{asn } Q)$

**Proof**

**References**


