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

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*: $A\ |\cap|\ B = \{||\} \longleftrightarrow (\forall x.\ x\ |\in|\ A \longrightarrow x\ |\notin|\ B)$
  $\langle proof \rangle$

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

type-notation *fmap* (**infix** $\langle \rightharpoonup_f \rangle$ *9*)

2.1 Difference

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

lift-definition
  $fmap\text{-}diff :: ('k \rightharpoonup_f 'v) \Rightarrow 'k\ fset \Rightarrow ('k \rightharpoonup_f 'v)$ (**infix** $\langle \ominus \rangle$ *110*)
is *map-diff*
$\langle proof \rangle$

2.2 Comprehension

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

lemma *make-map-transfer*[*transfer-rule*]: (*rel-fset* (=) ===> *A* ===> *rel-map* *A*) *make-map make-map*
⟨*proof*⟩

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

lift-definition
  make-fmap :: *'k fset* ⇒ *'v* ⇒ (*'k* $\rightharpoonup_f$ *'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* |∩| *fmdom B* = {||}
  shows *A* $++_f$ *B* = *B* $++_f$ *A*
⟨*proof*⟩ **including** *fset.lifting*
⟨*proof*⟩

lemma *make-fmap-union*:
  [*xs* |=> *v*] $++_f$ [*ys* |=> *v*] = [*xs* |∪| *ys* |=> *v*]
⟨*proof*⟩

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

⟨*proof*⟩

2.4 Lookup

lift-definition
  lookup :: (*'k* $\rightharpoonup_f$ *'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* |∈| *fmdom ys*
  shows *lookup* (*xs* $++_f$ *ys*) *k* = *lookup ys k*

⟨*proof*⟩ **including** *fset.lifting*
⟨*proof*⟩

lemma *lookup-union2*:
  assumes $k \mathrel{|\notin|} \mathit{fmdom}\ \mathit{ys}$
  shows $\mathit{lookup}\ (\mathit{xs} +\!+_f\ \mathit{ys})\ k = \mathit{lookup}\ \mathit{xs}\ k$
⟨*proof*⟩ **including** *fset.lifting*
⟨*proof*⟩

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

lemma *disjI* [*intro*]:
  assumes $\neg\ P \Longrightarrow Q$
  shows $P \vee Q$
⟨*proof*⟩

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

3.1 Projection functions on triples

definition $\mathit{fst3} :: {'a} \times {'b} \times {'c} \Rightarrow {'a}$
where $\mathit{fst3} \equiv \mathit{fst}$

```
definition snd3 :: 'a × 'b × 'c ⇒ 'b
where snd3 ≡ fst ∘ snd

definition thd3 :: 'a × 'b × 'c ⇒ 'c
where thd3 ≡ snd ∘ snd

lemma fst3-simp:
  ⋀a b c. fst3 (a,b,c) = a
⟨proof⟩

lemma snd3-simp:
  ⋀a b c. snd3 (a,b,c) = b
⟨proof⟩

lemma thd3-simp:
  ⋀a b c. thd3 (a,b,c) = c
⟨proof⟩

lemma tripleI:
  fixes T U
  assumes fst3 T = fst3 U
      and snd3 T = snd3 U
      and thd3 T = thd3 U
  shows T = U
⟨proof⟩

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* ⦄ · *proof* · ⦃ *chain* ⦄ · *might* · ⦃ *look* ⦄ · *like* · ⦃ *this* ⦄.

4.1 Projections

Project first assertion.

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

Project final assertion.

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

Project list of commands.

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

4.2 Chain length

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

lemma *len-comlist-chainlen*:
  length (*comlist* Π) = *chainlen* Π
⟨*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* Π.

fun
  nthtriple :: $('a,'c)$ *chain* $\Rightarrow$ *nat* $\Rightarrow$ $('a * 'c * 'a)$
where
  nthtriple $(\{\!|\ P\ |\!\} \cdot x \cdot \Pi)$ *0* = (P, x, *pre* Π)
| *nthtriple* $(\{\!|\ P\ |\!\} \cdot x \cdot \Pi)$ (*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* $\Pi \Longrightarrow$ *snd3* (*nthtriple* Π i) = (*comlist* Π)!i
⟨*proof*⟩

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
  $\mathit{chain\text{-}all}\ \varphi\ \{\!|\ \sigma\ |\!\} = \mathit{True}$
$|\ \mathit{chain\text{-}all}\ \varphi\ (\{\!|\ \sigma\ |\!\} \cdot x \cdot \Pi) = (\varphi\ (\sigma, x, \mathit{pre}\ \Pi) \wedge \mathit{chain\text{-}all}\ \varphi\ \Pi)$

lemma *chain-all-mono* [*mono*]:
  $x \leq y \Longrightarrow \mathit{chain\text{-}all}\ x \leq \mathit{chain\text{-}all}\ y$
⟨*proof*⟩

lemma *chain-all-nthtriple*:
  $(\mathit{chain\text{-}all}\ \varphi\ \Pi) = (\forall\, i < \mathit{chainlen}\ \Pi.\ \varphi\ (\mathit{nthtriple}\ \Pi\ i))$
⟨*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
  $\mathit{chainmap} :: ('a \Rightarrow 'b) \Rightarrow ('c \Rightarrow 'd) \Rightarrow ('a,'c)\ \mathit{chain} \Rightarrow ('b,'d)\ \mathit{chain}$
where
  $\mathit{chainmap}\ f\ g\ \{\!|\ P\ |\!\} = \{\!|\ f\ P\ |\!\}$
$|\ \mathit{chainmap}\ f\ g\ (\{\!|\ P\ |\!\} \cdot x \cdot \Pi) = \{\!|\ f\ P\ |\!\} \cdot g\ x \cdot \mathit{chainmap}\ f\ g\ \Pi$

Mapping over a chain preserves its length.

lemma *chainmap-preserves-length*:
  $\mathit{chainlen}\ (\mathit{chainmap}\ f\ g\ \Pi) = \mathit{chainlen}\ \Pi$
⟨*proof*⟩

lemma *pre-chainmap*:
  $\mathit{pre}\ (\mathit{chainmap}\ f\ g\ \Pi) = f\ (\mathit{pre}\ \Pi)$
⟨*proof*⟩

lemma *post-chainmap*:
  $\mathit{post}\ (\mathit{chainmap}\ f\ g\ \Pi) = f\ (\mathit{post}\ \Pi)$
⟨*proof*⟩

lemma *nthtriple-chainmap*:
  assumes $i < \mathit{chainlen}\ \Pi$
  shows $\mathit{nthtriple}\ (\mathit{chainmap}\ f\ g\ \Pi)\ i$
    $= (\lambda t.\ (f\ (\mathit{fst3}\ t),\ g\ (\mathit{snd3}\ t),\ f\ (\mathit{thd3}\ t)))\ (\mathit{nthtriple}\ \Pi\ i)$
⟨*proof*⟩

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 τ = $\{\!|\ \sigma\ |\!\} \cdot y \cdot \{\!|\ \tau\ |\!\}$
| $cSnoc$ $(\{\!|\ \sigma\ |\!\} \cdot x \cdot \Pi)$ y τ = $\{\!|\ \sigma\ |\!\} \cdot x \cdot (cSnoc\ \Pi\ y\ \tau)$

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

lemma *pre-snoc*:
pre $(cSnoc\ \Pi\ x\ P) = pre\ \Pi$
⟨*proof*⟩

lemma *post-snoc*:
post $(cSnoc\ \Pi\ x\ P) = P$
⟨*proof*⟩

lemma *comlist-snoc*:
comlist $(cSnoc\ \Pi\ x\ p) = comlist\ \Pi$ @ $[x]$
⟨*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 ∪ 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)
⟨proof⟩

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)
⟨proof⟩

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)
⟨proof⟩

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*:
  $[\![\ P \simeq Q\ ;\ P' \simeq Q'\ ]\!] \Longrightarrow P \otimes_c P' \simeq Q \otimes_c Q'$
⟨*proof*⟩

quotient-type *interface* = *conc-interface* / *equiv-int*
⟨*proof*⟩

lift-definition
  Ribbon :: *assertion* ⇒ *interface*
is *Ribbon-conc* ⟨*proof*⟩

lift-definition
  Emp-int :: *interface* (‹ε›)
is ε_c ⟨*proof*⟩

lift-definition
  Exists-int :: *string* ⇒ *interface* ⇒ *interface*
is *Exists-int-conc*
⟨*proof*⟩

lift-definition
  HComp-int :: *interface* ⇒ *interface* ⇒ *interface* (**infix** ‹⊗› *50*)
is *HComp-int-conc* ⟨*proof*⟩

lemma *hcomp-comm*:
  $(P \otimes Q) = (Q \otimes P)$
⟨*proof*⟩

lemma *hcomp-assoc*:
  $(P \otimes (Q \otimes R)) = ((P \otimes Q) \otimes R)$
⟨*proof*⟩

lemma *emp-hcomp*:
  $\varepsilon \otimes P = P$
⟨*proof*⟩

lemma *hcomp-emp*:
  $P \otimes \varepsilon = P$
⟨*proof*⟩

lemma *comp-fun-commute-hcomp*:
  comp-fun-commute (⊗)
⟨*proof*⟩

6.2 An iterated horizontal-composition operator

definition *iter-hcomp* :: (*'a fset*) ⇒ (*'a* ⇒ *interface*) ⇒ *interface*
where
  iter-hcomp X f ≡ *ffold* ((⊗) ∘ *f*) ε *X*

syntax *iter-hcomp-syntax* ::
  'a ⇒ (*'a fset*) ⇒ (*'a* ⇒ *interface*) ⇒ *interface*
    (‹(⨂ -|∈|-. -)› [*0*,*0*,*10*] *10*)
syntax-consts *iter-hcomp-syntax* == *iter-hcomp*

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

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 $\forall\ v \in \mathit{fset}\ vs.\ \varphi\ v = \varphi'\ v$
  shows $(\bigotimes v|{\in}|vs.\ \varphi\ v) = (\bigotimes v|{\in}|vs.\ \varphi'\ v)$
⟨*proof*⟩

lemma *iter-hcomp-empty*:
  shows $(\bigotimes x\ |{\in}|\ \{||\}.\ p\ x) = \varepsilon$
  ⟨*proof*⟩

lemma *iter-hcomp-insert*:
  assumes $v\ |{\notin}|\ ws$
  shows $(\bigotimes x\ |{\in}|\ \mathit{finsert}\ v\ ws.\ p\ x) = (p\ v \otimes (\bigotimes x\ |{\in}|\ ws.\ p\ x))$
⟨*proof*⟩

lemma *iter-hcomp-union*:
  assumes $vs\ |{\cap}|\ ws = \{||\}$
  shows $(\bigotimes x\ |{\in}|\ vs\ |{\cup}|\ ws.\ p\ x) = ((\bigotimes x\ |{\in}|\ vs.\ p\ x) \otimes (\bigotimes x\ |{\in}|\ ws.\ p\ x))$
⟨*proof*⟩

6.3 Semantics of interfaces

The semantics of an interface is an assertion.

fun
  $\mathit{conc\text{-}asn} :: \mathit{conc\text{-}interface} \Rightarrow \mathit{assertion}$
where
  $\mathit{conc\text{-}asn}\ (\mathit{Ribbon\text{-}conc}\ p) = p$
| $\mathit{conc\text{-}asn}\ (P \otimes_c Q) = (\mathit{conc\text{-}asn}\ P) \star (\mathit{conc\text{-}asn}\ Q)$
| $\mathit{conc\text{-}asn}\ (\varepsilon_c) = \mathit{Emp}$
| $\mathit{conc\text{-}asn}\ (\mathit{Exists\text{-}int\text{-}conc}\ x\ P) = \mathit{Exists}\ x\ (\mathit{conc\text{-}asn}\ P)$

lift-definition
  $\mathit{asn} :: \mathit{interface} \Rightarrow \mathit{assertion}$
is *conc-asn*
⟨*proof*⟩

lemma *asn-simps* [*simp*]:
  $\mathit{asn}\ (\mathit{Ribbon}\ p) = p$
  $\mathit{asn}\ (P \otimes Q) = (\mathit{asn}\ P) \star (\mathit{asn}\ Q)$
  $\mathit{asn}\ \varepsilon = \mathit{Emp}$
  $\mathit{asn}\ (\mathit{Exists\text{-}int}\ x\ P) = \mathit{Exists}\ x\ (\mathit{asn}\ P)$
⟨*proof*⟩

6.4 Program variables mentioned in an interface.

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

lift-definition
  $\mathit{rd\text{-}int} :: \mathit{interface} \Rightarrow \mathit{string\ set}$
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*:
  $\mathit{rd\text{-}ass}\ (\mathit{asn}\ P) = \mathit{rd\text{-}int}\ P$
⟨*proof*⟩

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

lemma *seq-fold*:
  $\bigwedge\Pi.\ [\![\ \mathit{length}\ \mathit{cs} = \mathit{chainlen}\ \Pi\ ;\ \mathit{p1} = \mathit{asn}\ (\mathit{pre}\ \Pi)\ ;\ \mathit{p2} = \mathit{asn}\ (\mathit{post}\ \Pi)\ ;$
  $\bigwedge i.\ i < \mathit{chainlen}\ \Pi \Longrightarrow \mathit{prov\text{-}triple}$
  $(\mathit{asn}\ (\mathit{fst3}\ (\mathit{nthtriple}\ \Pi\ i)),\ \mathit{cs}\ !\ i,\ \mathit{asn}\ (\mathit{thd3}\ (\mathit{nthtriple}\ \Pi\ i)))\ ]\!]$
  $\Longrightarrow \mathit{prov\text{-}triple}\ (\mathit{p1},\ \mathit{foldr}\ (;;)\ \mathit{cs}\ \mathit{Skip},\ \mathit{p2})$
⟨*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 $\mathit{sdiagram} = \mathit{SDiagram}\ (\mathit{cell} \times \mathit{interface})\ \mathit{list}$
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*) = (⋃ *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**

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

7.3 Soundness

lemma $soundness\text{-}strat\text{-}helper$:
$(prov\text{-}sdia\ D\ P\ Q \longrightarrow prov\text{-}triple\ (asn\ P,\ com\text{-}sdia\ D,\ asn\ Q)) \wedge$
$(prov\text{-}row\ \varrho\ P\ Q \longrightarrow prov\text{-}triple\ (asn\ P,\ com\text{-}cell\ (fst\ \varrho),\ asn\ Q)) \wedge$
$(prov\text{-}cell\ \gamma\ P\ Q \longrightarrow prov\text{-}triple\ (asn\ P,\ com\text{-}cell\ \gamma,\ asn\ Q))$
$\langle proof \rangle$

corollary $soundness\text{-}strat$:
assumes $prov\text{-}sdia\ D\ P\ Q$
shows $prov\text{-}triple\ (asn\ P,\ com\text{-}sdia\ D,\ asn\ Q)$
$\langle proof \rangle$

end

8 Syntax and proof rules for graphical diagrams

theory *Ribbons-Graphical* **imports**
Ribbons-Interfaces
begin

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

8.1 Syntax of graphical diagrams

Fix a type for node identifiers

typedecl *node*

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

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

Projecting components from a graph

fun $vertices :: diagram \Rightarrow node\ fset$ (‹-^*V*› [*1000*] *1000*)
where $(Graph\ V\ \Lambda\ E)\hat{}V = V$

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

fun $labelling :: diagram \Rightarrow labelling$ (‹-^Λ› [*1000*] *1000*)
where $(Graph\ V\ \Lambda\ E)\hat{}\Lambda = \Lambda$

fun $edges :: diagram \Rightarrow edge\ list$ (‹-^*E*› [*1000*] *1000*)
where $(Graph\ V\ \Lambda\ E)\hat{}E = E$

8.2 Well formedness of graphical diagrams

definition $acyclicity :: edge\ list \Rightarrow bool$
where
  $acyclicity\ E \equiv acyclic\ (\bigcup e \in set\ E.\ fset\ (fst3\ e) \times fset\ (thd3\ e))$

definition $linearity :: edge\ list \Rightarrow bool$
where
  $linearity\ E \equiv$
    $distinct\ E \wedge (\forall e \in set\ E.\ \forall f \in set\ E.\ e \neq f \longrightarrow$
    $fst3\ e\ |\cap|\ fst3\ f = \{||\} \wedge$
    $thd3\ e\ |\cap|\ thd3\ f = \{||\})$

lemma *linearityD*:
  assumes *linearity E*
  shows *distinct E*
  and $\bigwedge e\ f.\ [\![\ e \in set\ E\ ;\ f \in set\ E\ ;\ e \neq f\ ]\!] \Longrightarrow$
    $fst3\ e\ |\cap|\ fst3\ f = \{||\} \wedge$
    $thd3\ e\ |\cap|\ thd3\ f = \{||\}$
⟨*proof*⟩

lemma *linearityD2*:
  $linearity\ E \Longrightarrow (\forall e\ f.\ e \in set\ E \wedge f \in set\ E \wedge e \neq f \longrightarrow$

$\quad$ *fst3 e* |∩| *fst3 f* = {||} ∧
$\quad$ *thd3 e* |∩| *thd3 f* = {||})
⟨*proof*⟩

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 ∀ *v* ∈ *fset V*. *wf-ass* (Λ *v*)
    and ∀ *e* ∈ *set E*. *wf-com* (*snd3 e*)
    and *acyclicity E*
    and *linearity E*
    and ∀ *e* ∈ *set E*. *fst3 e* |∪| *thd3 e* |⊆| *V*
⟨*proof*⟩

8.3 Initial and terminal nodes

definition
  initials :: *diagram* ⇒ *node fset*
where
  initials G = *ffilter* (λ*v*. (∀ *e* ∈ *set G^E*. *v* |∉| *thd3 e*)) *G^V*

definition
  terminals :: *diagram* ⇒ *node fset*
where
  terminals G = *ffilter* (λ*v*. (∀ *e* ∈ *set G^E*. *v* |∉| *fst3 e*)) *G^V*

lemma *no-edges-imp-all-nodes-initial*:
  initials (*Graph V* Λ []) = *V*
⟨*proof*⟩

lemma *no-edges-imp-all-nodes-terminal*:
  terminals (*Graph V* Λ []) = *V*
⟨*proof*⟩

lemma *initials-in-vertices*:
  initials G $|\subseteq|$ *G^V*
$\langle$*proof*$\rangle$

lemma *terminals-in-vertices*:
  terminals G $|\subseteq|$ *G^V*
$\langle$*proof*$\rangle$

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* $\Rightarrow$ *interface* **and**
  bot-dia :: *diagram* $\Rightarrow$ *interface*
where
  bot-dia (*Graph V* Λ *E*) = ($\bigotimes$ *v* $|\in|$ *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*] $\Rightarrow$ *bool* **and**
  prov-com :: [*command-gadget*, *interface*, *interface*] $\Rightarrow$ *bool* **and**
  prov-ass :: *assertion-gadget* $\Rightarrow$ *bool*
where
  Skip: *prov-ass* (*Rib p*)
| *Exists*: *prov-dia G - -* $\Longrightarrow$ *prov-ass* (*Exists-dia x G*)
| *Basic*: *prov-triple* (*asn P*, *c*, *asn Q*) $\Longrightarrow$ *prov-com* (*Com c*) *P Q*
| *Choice*: $[\![$ *prov-dia G P Q* ; *prov-dia H P Q* $]\!]$
    $\Longrightarrow$ *prov-com* (*Choose-dia G H*) *P Q*
| *Loop*: *prov-dia G P P* $\Longrightarrow$ *prov-com* (*Loop-dia G*) *P P*
| *Main*: $[\![$ *wf-dia G* ; $\bigwedge$*v*. *v* $\in$ *fset G^V* $\Longrightarrow$ *prov-ass* (*G^*Λ *v*);
    $\bigwedge$*e*. *e* $\in$ *set G^E* $\Longrightarrow$ *prov-com* (*snd3 e*)
      ($\bigotimes$ *v* $|\in|$ *fst3 e*. *bot-ass* (*G^*Λ *v*))
      ($\bigotimes$ *v* $|\in|$ *thd3 e*. *top-ass* (*G^*Λ *v*))$]\!]$
    $\Longrightarrow$ *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)
 ∧ (*set* π = (*fset G^V*) <+> (*set G^E*))
 ∧ (∀ *i j v e*. *i* < *length* π ∧ *j* < *length* π ∧ π!*i* = *Inl v* ∧ π!*j* = *Inr e*
  ∧ *v* |∈| *fst3 e* ⟶ *i*<*j*)
 ∧ (∀ *j k w e*. *j* < *length* π ∧ *k* < *length* π ∧ π!*j* = *Inr e* ∧ π!*k* = *Inl w*
  ∧ *w* |∈| *thd3 e* ⟶ *j*<*k*) }

lemma *linsD*:
 assumes π ∈ *lins G*
 shows (*distinct* π)
  and (*set* π = (*fset G^V*) <+> (*set G^E*))
  and (∀ *i j v e*. *i* < *length* π ∧ *j* < *length* π
   ∧ π!*i* = *Inl v* ∧ π!*j* = *Inr e* ∧ *v* |∈| *fst3 e* ⟶ *i*<*j*)
  and (∀ *j k w e*. *j* < *length* π ∧ *k* < *length* π
   ∧ π!*j* = *Inr e* ∧ π!*k* = *Inl w* ∧ *w* |∈| *thd3 e* ⟶ *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 *coms-main* rule can be rewritten in a form that is more verbose but easier to prove monotonic.

lemma *coms-mono-helper*:
 (∀ *i*<*length* π. *case-sum* (*coms-ass* ∘ Λ) (*coms-com* ∘ *snd3*) (π!*i*) (*cs*!*i*))
 =
 ((∀ *i*. *i*<*length* π ∧ (∃ *v*. (π!*i*) = *Inl v*) ⟶
  coms-ass (Λ (*projl* (π!*i*))) (*cs*!*i*)) ∧
 (∀ *i*. *i*<*length* π ∧ (∃ *e*. (π!*i*) = *Inr e*) ⟶
  coms-com (*snd3* (*projr* (π!*i*))) (*cs*!*i*)))
⟨*proof*⟩

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] ⇒ 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.

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

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* ($\lambda\Pi$ 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*:

$\mathit{mk\text{-}ps\text{-}chain}\ \Pi\ (\pi\ @\ [x])$
$\quad = \mathit{cSnoc}\ (\mathit{mk\text{-}ps\text{-}chain}\ \Pi\ \pi)\ x\ (\mathit{next\text{-}ps}\ (\mathit{post}\ (\mathit{mk\text{-}ps\text{-}chain}\ \Pi\ \pi))\ x)$
$\langle \mathit{proof} \rangle$

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

lemma *mk-ps-chain-ccons*:
fixes $\pi\ \Pi$
shows $\mathit{mk\text{-}ps\text{-}chain}\ (\{\!\mid \sigma \mid\!\} \cdot x \cdot \Pi)\ \pi = \{\!\mid \sigma \mid\!\} \cdot x \cdot \mathit{mk\text{-}ps\text{-}chain}\ \Pi\ \pi$
$\langle \mathit{proof} \rangle$

lemma *pre-mk-ps-chain*:
fixes $\Pi\ \pi$
shows $\mathit{pre}\ (\mathit{mk\text{-}ps\text{-}chain}\ \Pi\ \pi) = \mathit{pre}\ \Pi$
$\langle \mathit{proof} \rangle$

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*:
$\mathit{comlist}\ (\mathit{mk\text{-}ps\text{-}chain}\ \Pi\ \pi) = \mathit{comlist}\ \Pi\ @\ \pi$
$\langle \mathit{proof} \rangle$

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

lemma *lins2D*:
  assumes $\pi \in$ *lins2 S G*
  shows *distinct* π
    and *set* π = (*fset G^V* − *fset S*) <+> *set G^E*
    and $\bigwedge$*i j v e*. ⟦ *i* < *length* π ; *j* < *length* π ;
      π!*i* = *Inl v* ; π!*j* = *Inr e* ; *v* |∈| *fst3 e* ⟧ ⟹ *i*<*j*
    and $\bigwedge$*i k w e*. ⟦ *j* < *length* π ; *k* < *length* π ;
      π!*j* = *Inr e* ; π!*k* = *Inl w* ; *w* |∈| *thd3 e* ⟧ ⟹ *j*<*k*
⟨*proof*⟩

lemma *lins2I*:
  assumes *distinct* π
    and *set* π = (*fset G^V* − *fset S*) <+> *set G^E*
    and $\bigwedge$*i j v e*. ⟦ *i* < *length* π ; *j* < *length* π ;
      π!*i* = *Inl v* ; π!*j* = *Inr e* ; *v* |∈| *fst3 e* ⟧ ⟹ *i*<*j*
    and $\bigwedge$*j k w e*. ⟦ *j* < *length* π ; *k* < *length* π ;
      π!*j* = *Inr e* ; π!*k* = *Inl w* ; *w* |∈| *thd3 e* ⟧ ⟹ *j*<*k*
  shows $\pi \in$ *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-chains G ≡ *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*] ⇒ *ps-chain set*
where
  ps-chains2 S G ≡ *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* {||}
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)

⟨*proof*⟩

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*]
⟨*proof*⟩

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* − *fst3 e*) *G′* =
  initial-ps2 S G ⊖ *fst3 e* $++_f$ [*thd3 e* |=> *Top*]
⟨*proof*⟩

lemma *next-lins2-vertex*:
  assumes *Inl v* # π ∈ *lins2 S G*
  assumes *v* |∉| *S*
  shows π ∈ *lins2* ({|*v*|} |∪| *S*) *G*
⟨*proof*⟩

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*⟩

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* |⊆| *initials G*
    and *wf-dia G*
    and Π ∈ *ps-chains2 S G*
    and *fcard G^V* + *length G^E* = *k* + *fcard S*
  shows *wf-ps-chain* Π ∧ (*post* Π = [*terminals G* |=> *Bot*])

$\langle proof \rangle$

corollary *wf-chains*:
  assumes *wf-dia G*
  assumes $\Pi \in$ *ps-chains G*
  shows *wf-ps-chain* $\Pi \wedge$ *post* Π = [*terminals G* |=> *Bot*]
$\langle proof \rangle$

9.2 Interface chains

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

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

The following functions convert a proofstate chain into an interface chain.

definition
  ps-to-int :: [*diagram*, *proofstate*] $\Rightarrow$ *interface*
where
  ps-to-int G $\sigma \equiv$
    $\bigotimes$ *v* |∈| *fmdom* σ. *case-topbot top-ass bot-ass* (*lookup* σ *v*) (G^Λ *v*)

definition
  ps-chain-to-int-chain :: [*diagram*, *ps-chain*] $\Rightarrow$ *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* Λ *E*) Π =
    chainmap (*ps-to-int* (*Graph V* Λ *E*)) ((*case-sum* (*Inl* $\circ$ Λ) (*Inr* $\circ$ *snd3*))) Π
$\langle proof \rangle$

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: ⋀c. wr-com c = {}
  shows
    (prov-dia G P Q ⟶
      (P = top-dia G ∧ Q = bot-dia G ∧
      (∀ c. coms-dia G c ⟶ prov-triple (asn P, c, asn Q))))
  ∧ (prov-com C P Q ⟶
      (∀ c. coms-com C c ⟶ prov-triple (asn P, c, asn Q)))
  ∧ (prov-ass A ⟶
      (∀ c. coms-ass A c ⟶ prov-triple (asn (top-ass A), c, asn (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 ⋀c. wr-com c = {}
  assumes prov-dia G P Q
  shows ∀ c. coms-dia G c ⟶ prov-triple (asn P, c, asn Q)
⟨proof⟩

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.