

Ribbon Proofs for Separation Logic (Isabelle Formalisation)

John Wickerson

February 6, 2026

Abstract

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

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

Contents

1	Introduction	2
2	Finite partial functions	3
2.1	Difference	3
2.2	Comprehension	3
2.3	Domain	4
2.4	Lookup	4
3	General purpose definitions and lemmas	5
3.1	Projection functions on triples	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

5	Assertions, commands, and separation logic proof rules	9
5.1	Assertions	9
5.2	Commands	10
5.3	Separation logic proof rules	11
6	Ribbon proof interfaces	12
6.1	Syntax of interfaces	12
6.2	An iterated horizontal-composition operator	13
6.3	Semantics of interfaces	14
6.4	Program variables mentioned in an interface.	15
7	Syntax and proof rules for stratified diagrams	15
7.1	Syntax of stratified diagrams	15
7.2	Proof rules for stratified diagrams	16
7.3	Soundness	17
8	Syntax and proof rules for graphical diagrams	17
8.1	Syntax of graphical diagrams	17
8.2	Well formedness of graphical diagrams	18
8.3	Initial and terminal nodes	19
8.4	Top and bottom interfaces	20
8.5	Proof rules for graphical diagrams	20
8.6	Extracting commands from diagrams	21
9	Soundness for graphical diagrams	22
9.1	Proofstate chains	23
9.2	Interface chains	28
9.3	Soundness proof	28

1 Introduction

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

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

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

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

2 Finite partial functions

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

```
lemma fdisjoint-iff:  $A \mid\cap\mid B = \{\mid\}$   $\longleftrightarrow (\forall x. x \mid\in\mid A \longrightarrow x \mid\notin\mid B)$ 
  <proof>
```

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

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

2.1 Difference

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

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

2.2 Comprehension

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

lemma *make-map-transfer*[*transfer-rule*]: $(rel\text{-}fset (=) \implies A \implies rel\text{-}map A)$ *make-map make-map*
 ⟨*proof*⟩

lemma *dom-make-map*:
 $dom (make\text{-}map\ ks\ v) = fset\ ks$
 ⟨*proof*⟩

lift-definition
 $make\text{-}fmap :: 'k\ fset \Rightarrow 'v \Rightarrow ('k \rightarrow_f 'v) (\langle [- \mid \Rightarrow -] \rangle)$
is *make-map parametric make-map-transfer*
 ⟨*proof*⟩

lemma *make-fmap-empty*[*simp*]: $[\{\} \mid \Rightarrow f] = fmapempty$
 ⟨*proof*⟩

2.3 Domain

lemma *fmap-add-commute*:
assumes $fmdom\ A \mid \cap \mid fmdom\ B = \{\}$
shows $A\ ++_f\ B = B\ ++_f\ A$
 ⟨*proof*⟩ **including** *fset.lifting*
 ⟨*proof*⟩

lemma *make-fmap-union*:
 $[xs \mid \Rightarrow v]\ ++_f\ [ys \mid \Rightarrow v] = [xs \mid \cup \mid ys \mid \Rightarrow v]$
 ⟨*proof*⟩

lemma *fdom-make-fmap*: $fmdom\ [ks \mid \Rightarrow v] = ks$
 ⟨*proof*⟩

2.4 Lookup

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

lemma *lookup-make-fmap*:
assumes $k \in fset\ ks$
shows $lookup\ [ks \mid \Rightarrow v]\ k = v$
 ⟨*proof*⟩

lemma *lookup-make-fmap1*:
 $lookup\ [\{ | k | \} \mid \Rightarrow v]\ k = v$
 ⟨*proof*⟩

lemma *lookup-union1*:
assumes $k \in \mid fmdom\ ys$
shows $lookup\ (xs\ ++_f\ ys)\ k = lookup\ ys\ k$

$\langle proof \rangle$ **including** *fset.lifting*
 $\langle proof \rangle$

lemma *lookup-union2*:
 assumes $k \notin \text{fmdom } ys$
 shows $\text{lookup } (xs ++_f ys) k = \text{lookup } xs k$
 $\langle proof \rangle$ **including** *fset.lifting*
 $\langle proof \rangle$

lemma *lookup-union3*:
 assumes $k \notin \text{fmdom } xs$
 shows $\text{lookup } (xs ++_f ys) k = \text{lookup } ys k$
 $\langle proof \rangle$ **including** *fset.lifting*
 $\langle proof \rangle$

end

3 General purpose definitions and lemmas

theory *JHelper* **imports**

Main

begin

lemma *Collect-iff*:
 $a \in \{x \mid P x\} \equiv P a$
 $\langle proof \rangle$

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

lemma *nth-in-set*:
 $\llbracket i < \text{length } xs \ ; \ xs ! i = x \rrbracket \Longrightarrow x \in \text{set } xs$ $\langle proof \rangle$

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

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

3.1 Projection functions on triples

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

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

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

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

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

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

lemma $tripleI$:
fixes $T\ U$
assumes $fst3\ T = fst3\ U$
and $snd3\ T = snd3\ U$
and $thd3\ T = thd3\ U$
shows $T = U$
 $\langle proof \rangle$

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 $(a,c)\ chain =$
 $cNil\ a \quad (\langle \{ \} \rangle)$
 $| cCons\ a\ c\ (a,c)\ chain \quad (\langle \{ \} \cdot \dots \rightarrow [0,0,0]\ 60)$

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

4.1 Projections

Project first assertion.

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

Project final assertion.

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

Project list of commands.

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

4.2 Chain length

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

lemma *len-comlist-chainlen*:
 $length (comlist \Pi) = chainlen \Pi$
<proof>

4.3 Extracting triples from chains

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

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

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

lemma *snds-of-triples-form-comlist*:
fixes Πi
shows $i < chainlen \Pi \Longrightarrow snd3 (nthtriple \Pi i) = (comlist \Pi)!i$
<proof>

4.4 Evaluating a predicate on each triple of a chain

$chain\text{-}all\ \varphi$ holds of Π iff φ holds for each of Π 's individual triples.

fun

$chain\text{-}all :: (('a \times 'c \times 'a) \Rightarrow bool) \Rightarrow ('a, 'c)\ chain \Rightarrow bool$

where

$chain\text{-}all\ \varphi\ \{\!\! \{\sigma\}\!\!\} = True$

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

lemma $chain\text{-}all\text{-}mono$ [*mono*]:

$x \leq y \Longrightarrow chain\text{-}all\ x \leq chain\text{-}all\ y$

<proof>

lemma $chain\text{-}all\text{-}nthtriple$:

$(chain\text{-}all\ \varphi\ \Pi) = (\forall i < chainlen\ \Pi. \varphi\ (nthtriple\ \Pi\ i))$

<proof>

4.5 A map function for proof chains

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

fun

$chainmap :: ('a \Rightarrow 'b) \Rightarrow ('c \Rightarrow 'd) \Rightarrow ('a, 'c)\ chain \Rightarrow ('b, 'd)\ chain$

where

$chainmap\ f\ g\ \{\!\! \{P\}\!\!\} = \{\!\! \{f\ P\}\!\!\}$

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

Mapping over a chain preserves its length.

lemma $chainmap\text{-}preserves\text{-}length$:

$chainlen\ (chainmap\ f\ g\ \Pi) = chainlen\ \Pi$

<proof>

lemma $pre\text{-}chainmap$:

$pre\ (chainmap\ f\ g\ \Pi) = f\ (pre\ \Pi)$

<proof>

lemma $post\text{-}chainmap$:

$post\ (chainmap\ f\ g\ \Pi) = f\ (post\ \Pi)$

<proof>

lemma $nthtriple\text{-}chainmap$:

assumes $i < chainlen\ \Pi$

shows $nthtriple\ (chainmap\ f\ g\ \Pi)\ i$

$= (\lambda t. (f\ (fst3\ t), g\ (snd3\ t), f\ (thd3\ t)))\ (nthtriple\ \Pi\ i)$

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

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

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

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

lemma *comlist-snoc*:
 $comlist (cSnoc \Pi x p) = comlist \Pi @ [x]$
 $\langle proof \rangle$

end

5 Assertions, commands, and separation logic proof rules

theory *Ribbons-Basic* **imports**
 $Main$
begin

We define a command language, assertions, and the rules of separation logic, plus some derived rules that are used by our tool. This is the only theory file that is loaded by the tool. We keep it as small as possible.

5.1 Assertions

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

typedecl *assertion*

axiomatization
 $Emp :: assertion$

axiomatization

Star :: *assertion* \Rightarrow *assertion* \Rightarrow *assertion* (**infixr** $\langle \star \rangle$ 55)
where
star-comm: $p \star q = q \star p$ **and**
star-assoc: $(p \star q) \star r = p \star (q \star r)$ **and**
star-emp: $p \star \text{Emp} = p$ **and**
emp-star: $\text{Emp} \star p = p$

lemma *star-rot*:
 $q \star p \star r = p \star q \star r$
 $\langle \text{proof} \rangle$

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

Extracting the set of program variables mentioned in an assertion.

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

5.2 Commands

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

typedecl *command*

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

axiomatization
Loop :: *command* \Rightarrow *command*

axiomatization
Skip :: *command*

axiomatization
Seq :: *command* \Rightarrow *command* \Rightarrow *command* (**infixr** $\langle ;; \rangle$ 55)
where *seq-assoc*: $c1 ;; (c2 ;; c3) = (c1 ;; c2) ;; c3$
and *seq-skip*: $c ;; \text{Skip} = c$
and *skip-seq*: $\text{Skip} ;; c = c$

Extracting the set of program variables read by a command.

axiomatization
rd-com :: *command* \Rightarrow *string set*
where *rd-com-choose*: $\text{rd-com } (\text{Choose } c1 c2) = \text{rd-com } c1 \cup \text{rd-com } c2$
and *rd-com-loop*: $\text{rd-com } (\text{Loop } c) = \text{rd-com } c$
and *rd-com-skip*: $\text{rd-com } (\text{Skip}) = \{\}$

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

exists: $prov-triple (p, c, q) \Longrightarrow prov-triple (Exists\ x\ p, c, Exists\ x\ q)$

| *choose*: $\llbracket prov-triple (p, c1, q); prov-triple (p, c2, q) \rrbracket$

$\Longrightarrow prov-triple (p, Choose\ c1\ c2, q)$

| *loop*: $prov-triple (p, c, p) \Longrightarrow prov-triple (p, Loop\ c, p)$

| *frame*: $\llbracket prov-triple (p, c, q); wr-com(c) \cap rd-ass(r) = \{\} \rrbracket$

$\Longrightarrow prov-triple (p \star r, c, q \star r)$

| *skip*: $prov-triple (p, Skip, p)$

| *seq*: $\llbracket prov-triple (p, c1, q); prov-triple (q, c2, r) \rrbracket$

$\Longrightarrow prov-triple (p, c1 ;; c2, r)$

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** $\langle \otimes_c \rangle$ 50)

| *Emp-int-conc* ($\langle \varepsilon_c \rangle$)

| *Exists-int-conc string conc-interface*

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

inductive

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

where

refl: $P \simeq P$

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

| *trans*: $[P \simeq Q; Q \simeq R] \Longrightarrow P \simeq R$

| *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*: $P \simeq Q \Longrightarrow \text{Exists-int-conc } x P \simeq \text{Exists-int-conc } x Q$

| *hcomp-conc-assoc*: $P \otimes_c (Q \otimes_c R) \simeq (P \otimes_c Q) \otimes_c R$

| *hcomp-conc-comm*: $P \otimes_c Q \simeq Q \otimes_c P$

| *hcomp-conc-unit1*: $\varepsilon_c \otimes_c P \simeq P$

| *hcomp-conc-unit2*: $P \otimes_c \varepsilon_c \simeq P$

lemma *equiv-int-cong-hcomp*:

$[P \simeq Q; P' \simeq Q'] \Longrightarrow P \otimes_c P' \simeq Q \otimes_c Q'$

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

lift-definition

Ribbon :: *assertion* \Rightarrow *interface*
is *Ribbon-conc* \langle proof \rangle

lift-definition

Emp-int :: *interface* $(\langle \varepsilon \rangle)$
is ε_c \langle proof \rangle

lift-definition

Exists-int :: *string* \Rightarrow *interface* \Rightarrow *interface*
is *Exists-int-conc*
 \langle proof \rangle

lift-definition

HComp-int :: *interface* \Rightarrow *interface* \Rightarrow *interface* (**infix** $\langle \otimes \rangle$ 50)
is *HComp-int-conc* \langle proof \rangle

lemma *hcomp-comm*:

$(P \otimes Q) = (Q \otimes P)$
 \langle proof \rangle

lemma *hcomp-assoc*:

$(P \otimes (Q \otimes R)) = ((P \otimes Q) \otimes R)$
 \langle proof \rangle

lemma *emp-hcomp*:

$\varepsilon \otimes P = P$
 \langle proof \rangle

lemma *hcomp-emp*:

$P \otimes \varepsilon = P$
 \langle proof \rangle

lemma *comp-fun-commute-hcomp*:

comp-fun-commute (\otimes)
 \langle proof \rangle

6.2 An iterated horizontal-composition operator

definition *iter-hcomp* :: $(\text{'a fset}) \Rightarrow (\text{'a} \Rightarrow \text{interface}) \Rightarrow \text{interface}$
where

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

syntax *iter-hcomp-syntax* ::

$\text{'a} \Rightarrow (\text{'a fset}) \Rightarrow (\text{'a} \Rightarrow \text{interface}) \Rightarrow \text{interface}$
 $(\langle \langle \otimes \rangle \text{-} \langle \text{infix} \text{-} \rangle [0,0,10] 10 \rangle)$

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

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

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

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

lemma *iter-hcomp-cong*:

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

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 $(\bigotimes x \in \text{finset } v ws. p x) = (p v \otimes (\bigotimes x \in ws. p x))$

<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

conc-asn :: *conc-interface* \Rightarrow *assertion*

where

conc-asn (*Ribbon-conc* p) = p

| *conc-asn* ($P \otimes_c Q$) = (*conc-asn* P) \star (*conc-asn* Q)

| *conc-asn* (ε_c) = *Emp*

| *conc-asn* (*Exists-int-conc* $x P$) = *Exists* x (*conc-asn* P)

lift-definition

asn :: *interface* \Rightarrow *assertion*

is *conc-asn*

<proof>

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

asn (*Ribbon* p) = p

asn ($P \otimes Q$) = (*asn* P) \star (*asn* Q)

asn ε = *Emp*

asn (*Exists-int* $x P$) = *Exists* x (*asn* P)

<proof>

6.4 Program variables mentioned in an interface.

```

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

```

lift-definition

```

  rd-int :: interface  $\Rightarrow$  string set
is rd-conc-int
<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:

```

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

```

end

7 Syntax and proof rules for stratified diagrams

theory Ribbons-Stratified imports

Ribbons-Interfaces

Proofchain

begin

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

7.1 Syntax of stratified diagrams

```

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

```

and cell =

Filler interface

| *Basic interface command interface*

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

datatype-compat *sdiagram cell*

type-synonym *row = cell × interface*

Extracting the command from a stratified diagram.

fun

com-sdia :: *sdiagram* ⇒ *command* **and**

com-cell :: *cell* ⇒ *command*

where

com-sdia (*SDiagram* ρs) = *foldr* (*;*) (*map* (*com-cell* ∘ *fst*) ρs) *Skip*

| *com-cell* (*Filler* *P*) = *Skip*

| *com-cell* (*Basic* *P c Q*) = *c*

| *com-cell* (*Exists-sdia* *x D*) = *com-sdia* *D*

| *com-cell* (*Choose-sdia* *P D E Q*) = *Choose* (*com-sdia* *D*) (*com-sdia* *E*)

| *com-cell* (*Loop-sdia* *P D Q*) = *Loop* (*com-sdia* *D*)

Extracting the program variables written by a stratified diagram.

fun

wr-sdia :: *sdiagram* ⇒ *string set* **and**

wr-cell :: *cell* ⇒ *string set*

where

wr-sdia (*SDiagram* ρs) = $\bigcup r \in \text{set } \rho s. \text{wr-cell } (\text{fst } r)$

| *wr-cell* (*Filler* *P*) = $\{\}$

| *wr-cell* (*Basic* *P c Q*) = *wr-com* *c*

| *wr-cell* (*Exists-sdia* *x D*) = *wr-sdia* *D*

| *wr-cell* (*Choose-sdia* *P D E Q*) = *wr-sdia* *D* ∪ *wr-sdia* *E*

| *wr-cell* (*Loop-sdia* *P D Q*) = *wr-sdia* *D*

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

lemma *wr-sdia-is-wr-com*:

fixes ρs :: *row list*

and ρ :: *row*

shows (*wr-sdia* *D* = *wr-com* (*com-sdia* *D*))

and (*wr-cell* γ = *wr-com* (*com-cell* γ))

and ($\bigcup \rho \in \text{set } \rho s. \text{wr-cell } (\text{fst } \rho)$)

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

```

7.3 Soundness

lemma *soundness-strat-helper*:

```

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

typedecl *node*

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

```

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

```

Projecting components from a graph

```

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

```

```

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

```

```

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

```

```

fun edges :: diagram  $\Rightarrow$  edge list ( $\langle$ - $\hat{E}$  $\rangle$  [1000] 1000)
where (Graph V  $\wedge$  E) $\hat{E}$  = E

```

8.2 Well formedness of graphical diagrams

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

where

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

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

where

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

lemma *linearityD*:

assumes *linearity* *E*

shows *distinct* *E*

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

$\text{fst3 } e \mid \cap \mid \text{fst3 } f = \{\mid\} \wedge$

$\text{thd3 } e \mid \cap \mid \text{thd3 } f = \{\mid\}$

\langle *proof* \rangle

lemma *linearityD2*:

linearity *E* \Longrightarrow ($\forall e f. e \in \text{set } E \wedge f \in \text{set } E \wedge e \neq f \longrightarrow$

$fst3\ e\ |\cap|\ fst3\ f = \{\|\} \wedge$
 $thd3\ e\ |\cap|\ thd3\ f = \{\|\}$
 <proof>

inductive

$wf-ass :: assertion-gadget \Rightarrow bool$ **and**
 $wf-com :: command-gadget \Rightarrow bool$ **and**
 $wf-dia :: diagram \Rightarrow bool$

where

$wf-rib: wf-ass\ (Rib\ p)$
 $| wf-exists: wf-dia\ G \Longrightarrow wf-ass\ (Exists-dia\ x\ G)$
 $| wf-com: wf-com\ (Com\ c)$
 $| wf-choice: \llbracket wf-dia\ G ; wf-dia\ H \rrbracket \Longrightarrow wf-com\ (Choose-dia\ G\ H)$
 $| wf-loop: wf-dia\ G \Longrightarrow wf-com\ (Loop-dia\ G)$
 $| wf-dia: \llbracket \forall e \in set\ E. wf-com\ (snd3\ e) ; \forall v \in fset\ V. wf-ass\ (\Lambda\ v) ;$
 $acyclicity\ E ; linearity\ E ; \forall e \in set\ E. fst3\ e\ |\cup|\ thd3\ e\ |\subseteq|\ V \rrbracket \Longrightarrow$
 $wf-dia\ (Graph\ V\ \Lambda\ E)$

inductive-cases $wf-dia-inv'$: $wf-dia\ (Graph\ V\ \Lambda\ E)$

lemma $wf-dia-inv$:

assumes $wf-dia\ (Graph\ V\ \Lambda\ E)$
shows $\forall v \in fset\ V. wf-ass\ (\Lambda\ v)$
and $\forall e \in set\ E. wf-com\ (snd3\ e)$
and $acyclicity\ E$
and $linearity\ E$
and $\forall e \in set\ E. fst3\ e\ |\cup|\ thd3\ e\ |\subseteq|\ V$
 <proof>

8.3 Initial and terminal nodes

definition

$initials :: diagram \Rightarrow node\ fset$

where

$initials\ G = ffilter\ (\lambda v. (\forall e \in set\ G^{\wedge}E. v\ |\notin|\ thd3\ e))\ G^{\wedge}V$

definition

$terminals :: diagram \Rightarrow node\ fset$

where

$terminals\ G = ffilter\ (\lambda v. (\forall e \in set\ G^{\wedge}E. v\ |\notin|\ fst3\ e))\ G^{\wedge}V$

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

$initials\ (Graph\ V\ \Lambda\ []) = V$
 <proof>

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

$terminals\ (Graph\ V\ \Lambda\ []) = V$
 <proof>

lemma *initials-in-vertices*:

initials $G \mid \subseteq \mid G \hat{\wedge} V$

<proof>

lemma *terminals-in-vertices*:

terminals $G \mid \subseteq \mid G \hat{\wedge} V$

<proof>

8.4 Top and bottom interfaces

primrec

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

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

where

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

\mid *top-ass* (*Rib* p) = *Ribbon* p

\mid *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* $V \Lambda E$) = $(\otimes v \mid \in \mid \text{terminals } (\text{Graph } V \Lambda E). \text{bot-ass } (\Lambda v))$

\mid *bot-ass* (*Rib* p) = *Ribbon* p

\mid *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)

\mid *Exists*: *prov-dia* $G \text{ - - } \Longrightarrow \text{prov-ass } (\text{Exists-dia } x G)$

\mid *Basic*: *prov-triple* (*asn* P , c , *asn* Q) $\Longrightarrow \text{prov-com } (\text{Com } c) P Q$

\mid *Choice*: $\llbracket \text{prov-dia } G P Q ; \text{prov-dia } H P Q \rrbracket$

$\Longrightarrow \text{prov-com } (\text{Choose-dia } G H) P Q$

\mid *Loop*: *prov-dia* $G P P \Longrightarrow \text{prov-com } (\text{Loop-dia } G) P P$

\mid *Main*: $\llbracket \text{wf-dia } G ; \bigwedge v. v \in \text{fset } G \hat{\wedge} V \Longrightarrow \text{prov-ass } (G \hat{\wedge} \Lambda v);$

$\bigwedge e. e \in \text{set } G \hat{\wedge} E \Longrightarrow \text{prov-com } (\text{snd3 } e)$

$(\otimes v \mid \in \mid \text{fst3 } e. \text{bot-ass } (G \hat{\wedge} \Lambda v))$

$(\otimes v \mid \in \mid \text{thd3 } e. \text{top-ass } (G \hat{\wedge} \Lambda v)) \rrbracket$

$\Longrightarrow \text{prov-dia } G (\text{top-dia } G) (\text{bot-dia } G)$

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

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 \Rightarrow lin set*

where

lins G \equiv { $\pi :: lin.$

(distinct π)

$\wedge (set\ \pi = (fset\ G^{\wedge}V) <+> (set\ G^{\wedge}E))$

$\wedge (\forall i\ j\ v\ e. i < length\ \pi \wedge j < length\ \pi \wedge \pi!i = Inl\ v \wedge \pi!j = Inr\ e$

$\wedge v\ |\in|\ fst3\ e \longrightarrow i < j)$

$\wedge (\forall j\ k\ w\ e. j < length\ \pi \wedge k < length\ \pi \wedge \pi!j = Inr\ e \wedge \pi!k = Inl\ w$

$\wedge w\ |\in|\ thd3\ e \longrightarrow j < k)$ }

lemma *linsD:*

assumes $\pi \in lins\ G$

shows *(distinct π)*

and $(set\ \pi = (fset\ G^{\wedge}V) <+> (set\ G^{\wedge}E))$

and $(\forall i\ j\ v\ e. i < length\ \pi \wedge j < length\ \pi$

$\wedge \pi!i = Inl\ v \wedge \pi!j = Inr\ e \wedge v\ |\in|\ fst3\ e \longrightarrow i < j)$

and $(\forall j\ k\ w\ e. j < length\ \pi \wedge k < length\ \pi$

$\wedge \pi!j = Inr\ e \wedge \pi!k = Inl\ w \wedge w\ |\in|\ thd3\ e \longrightarrow j < k)$

<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:*

$(\forall i < length\ \pi. case-sum\ (coms-ass \circ \Lambda)\ (coms-com \circ snd3)\ (\pi!i)\ (cs!i))$

$=$

$((\forall i. i < length\ \pi \wedge (\exists v. (\pi!i) = Inl\ v) \longrightarrow$

$coms-ass\ (\Lambda\ (projl\ (\pi!i)))\ (cs!i)) \wedge$

$(\forall i. i < length\ \pi \wedge (\exists e. (\pi!i) = Inr\ e) \longrightarrow$

$coms-com\ (snd3\ (projr\ (\pi!i)))\ (cs!i)))$

<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*: [$\pi \in \text{lins } (\text{Graph } V \ \Lambda \ E)$; $\text{length } cs = \text{length } \pi$;
 $\forall i < \text{length } \pi. \text{case-sum } (\text{coms-ass } \circ \Lambda) (\text{coms-com } \circ \text{snd}) (\pi!i) (cs!i)$]
⇒ *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* \rightarrow_f *topbot*

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

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

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

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

where

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

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

definition

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

where

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

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

fixes $\pi\ \Pi$

shows *chainlen* (*mk-ps-chain* $\Pi\ \pi$) = *chainlen* Π + *length* π
 $\langle proof \rangle$

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

lemma *mk-ps-chain-cons*:

mk-ps-chain Π ($x\ \#\ \pi$) = *mk-ps-chain* (*cSnoc* $\Pi\ x\ (next-ps\ (post\ \Pi)\ x)$) π
 $\langle proof \rangle$

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

lemma *mk-ps-chain-snoc*:

$mk-ps-chain \ \Pi \ (\pi \ @ \ [x])$
 $= cSnoc \ (mk-ps-chain \ \Pi \ \pi) \ x \ (next-ps \ (post \ (mk-ps-chain \ \Pi \ \pi)) \ x)$
 <proof>

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

lemma *mk-ps-chain-ccons*:

fixes $\pi \ \Pi$

shows $mk-ps-chain \ (\!\! \{ \sigma \} \cdot x \cdot \Pi) \ \pi = \!\! \{ \sigma \} \cdot x \cdot mk-ps-chain \ \Pi \ \pi$

<proof>

lemma *pre-mk-ps-chain*:

fixes $\Pi \ \pi$

shows $pre \ (mk-ps-chain \ \Pi \ \pi) = pre \ \Pi$

<proof>

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

lemma *comlist-mk-ps-chain*:

$comlist \ (mk-ps-chain \ \Pi \ \pi) = comlist \ \Pi \ @ \ \pi$

<proof>

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

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

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

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

where

$lins2 \ S \ G \equiv \{ \pi :: \ lin \ .$

(*distinct* π)

$\wedge \ (set \ \pi = (fset \ G \hat{\ } V - fset \ S) \ <+> \ set \ G \hat{\ } E)$

$\wedge \ (\forall \ i \ j \ v \ e. \ i < length \ \pi \wedge \ j < length \ \pi$

$\wedge \ \pi!i = Inl \ v \wedge \ \pi!j = Inr \ e \wedge \ v \ |\in| \ fst3 \ e \longrightarrow \ i < j)$

$\wedge \ (\forall \ j \ k \ w \ e. \ j < length \ \pi \wedge \ k < length \ \pi$

$\wedge \ \pi!j = Inr \ e \wedge \ \pi!k = Inl \ w \wedge \ w \ |\in| \ thd3 \ e \longrightarrow \ j < k) \}$

lemma *lins2D*:

assumes $\pi \in \text{lins2 } S \ G$

shows *distinct* π

and $\text{set } \pi = (\text{fset } G \hat{\ } V - \text{fset } S) \langle + \rangle \text{set } G \hat{\ } E$

and $\bigwedge i \ j \ v \ e. \llbracket i < \text{length } \pi ; j < \text{length } \pi ;$

$\pi!i = \text{Inl } v ; \pi!j = \text{Inr } e ; v \in | \text{fst3 } e \rrbracket \implies i < j$

and $\bigwedge i \ k \ w \ e. \llbracket j < \text{length } \pi ; k < \text{length } \pi ;$

$\pi!j = \text{Inr } e ; \pi!k = \text{Inl } w ; w \in | \text{thd3 } e \rrbracket \implies j < k$

<proof>

lemma *lins2I*:

assumes *distinct* π

and $\text{set } \pi = (\text{fset } G \hat{\ } V - \text{fset } S) \langle + \rangle \text{set } G \hat{\ } E$

and $\bigwedge i \ j \ v \ e. \llbracket i < \text{length } \pi ; j < \text{length } \pi ;$

$\pi!i = \text{Inl } v ; \pi!j = \text{Inr } e ; v \in | \text{fst3 } e \rrbracket \implies i < j$

and $\bigwedge j \ k \ w \ e. \llbracket j < \text{length } \pi ; k < \text{length } \pi ;$

$\pi!j = \text{Inr } e ; \pi!k = \text{Inl } w ; w \in | \text{thd3 } e \rrbracket \implies j < k$

shows $\pi \in \text{lins2 } S \ G$

<proof>

When S is empty, the two definitions coincide.

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

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

<proof>

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

definition

$\text{initial-ps} :: \text{diagram} \Rightarrow \text{proofstate}$

where

$\text{initial-ps } G \equiv [\text{initials } G \ | \Rightarrow \ \text{Top}]$

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

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

where

$\text{initial-ps2 } S \ G \equiv [\text{initials } G - S \ | \Rightarrow \ \text{Top}] \ ++_f [S \ | \Rightarrow \ \text{Bot}]$

When S is empty, the above two definitions coincide.

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

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

<proof>

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

definition

$\text{ps-chains} :: \text{diagram} \Rightarrow \text{ps-chain set}$

where

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

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

definition

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

where

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

When S is empty, the above two definitions coincide.

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

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

<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 $(\sigma \ ++_f \ disjoint \ [\{|v|\} \ |=> \ Top \], \ Inl \ v, \ \sigma \ ++_f \ disjoint \ [\{|v|\} \ |=> \ Bot \])$, where v is a node, or of the form $(\sigma \ ++_f \ disjoint \ [\{|vs|\} \ |=> \ Bot \])$, $Inr \ e, \ \sigma \ ++_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

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

where

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

lemma *wf-ps-triple-nodeI:*

$$\begin{aligned} \text{assumes } & \exists \sigma. \ v \notin \ fdom \ \sigma \ \wedge \\ & \ \sigma1 = [\{|v|\} \ |=> \ Top \] \ ++_f \ \sigma \ \wedge \\ & \ \sigma2 = [\{|v|\} \ |=> \ Bot \] \ ++_f \ \sigma \\ \text{shows } & \ wf\text{-ps-triple } (\sigma1, \ Inl \ v, \ \sigma2) \end{aligned}$$

<proof>

lemma *wf-ps-triple-edgeI:*

$$\begin{aligned} \text{assumes } & \exists \sigma. \ (fst3 \ e \ \cup \ thd3 \ e) \ \cap \ fdom \ \sigma = \{\|\} \\ & \ \wedge \ \sigma1 = [\fst3 \ e \ |=> \ Bot \] \ ++_f \ \sigma \\ & \ \wedge \ \sigma2 = [\thd3 \ e \ |=> \ Top \] \ ++_f \ \sigma \\ \text{shows } & \ wf\text{-ps-triple } (\sigma1, \ Inr \ e, \ \sigma2) \end{aligned}$$

<proof>

definition

wf-ps-chain :: *ps-chain* \Rightarrow *bool*

where

wf-ps-chain \equiv *chain-all wf-ps-triple*

lemma *next-initial-ps2-vertex*:

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

<proof>

lemma *next-initial-ps2-edge*:

assumes *G* = *Graph* *V* \wedge *E* **and** *G'* = *Graph* *V'* \wedge *E'* **and**
V' = *V* - *fst3* *e* **and** *E'* = *removeAll* *e* *E* **and** *e* \in *set* *E* **and**
fst3 *e* $\mid \subseteq \mid S$ **and** *S* $\mid \subseteq \mid$ *initials* *G* **and** *wf-dia* *G*
shows *initial-ps2* (*S* - *fst3* *e*) *G'* =
initial-ps2 *S G* \ominus *fst3* *e* ++_f [*thd3* *e* $\mid \Rightarrow Top$]

<proof>

lemma *next-lins2-vertex*:

assumes *Inl* *v* $\#$ $\pi \in$ *lins2* *S G*
assumes *v* $\notin \mid S$
shows $\pi \in$ *lins2* ($\{|v|\} \mid \cup \mid S$) *G*

<proof>

lemma *next-lins2-edge*:

assumes *Inr* *e* $\#$ $\pi \in$ *lins2* *S* (*Graph* *V* \wedge *E*)
and *vs* $\mid \subseteq \mid S$
and *e* = (*vs*, *c*, *ws*)
shows $\pi \in$ *lins2* (*S* - *vs*) (*Graph* (*V* - *vs*) \wedge (*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* $\mid \subseteq \mid$ *initials* *G*
and *wf-dia* *G*
and $\Pi \in$ *ps-chains2* *S G*
and *fcard* *G* \wedge *V* + *length* *G* \wedge *E* = *k* + *fcard* *S*
shows *wf-ps-chain* $\Pi \wedge$ (*post* $\Pi = [terminals$ *G* $\mid \Rightarrow Bot]$)

<proof>

corollary *wf-chains*:

assumes *wf-dia* G

assumes $\Pi \in \text{ps-chains } G$

shows *wf-ps-chain* $\Pi \wedge \text{post } \Pi = [\text{terminals } G \mid => \text{Bot}]$

<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

ps-to-int :: [*diagram*, *proofstate*] \Rightarrow *interface*

where

ps-to-int $G \sigma \equiv$

$\bigotimes v \mid \in \mid \text{fmdom } \sigma. \text{case-topbot top-ass bot-ass (lookup } \sigma \ v) (G \wedge \Lambda \ v)$

definition

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

where

ps-chain-to-int-chain $G \Pi \equiv$

$\text{chainmap (ps-to-int } G) ((\text{case-sum (Inl } \circ G \wedge \Lambda) (\text{Inr } \circ \text{snd3}))) \Pi$

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

ps-chain-to-int-chain (*Graph* $V \ \Lambda \ E$) $\Pi =$

$\text{chainmap (ps-to-int (Graph } V \ \Lambda \ E)) ((\text{case-sum (Inl } \circ \Lambda) (\text{Inr } \circ \text{snd3}))) \Pi$

<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 (*asn* P , c , *asn* Q) is provable for each command c that can be extracted from G ; (2) if a command-gadget C is provable w.r.t. interfaces P and Q , then the Hoare triple (*asn* P , c , *asn* Q) is provable for each command c that

can be extracted from C ; and (3) if an assertion-gadget A is provable, and if the top and bottom interfaces of A are P and Q respectively, then the Hoare triple $(asn\ P, c, asn\ Q)$ is provable for each command c that can be extracted from A .

lemma *soundness-graphical-helper*:

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

shows

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

$\langle proof \rangle$

The soundness theorem states that any diagram provable using the proof rules for ribbons can be recreated as a valid proof in separation logic.

corollary *soundness-graphical*:

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

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

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

$\langle proof \rangle$

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.