# Ribbon Proofs for Separation Logic (Isabelle Formalisation) 

John Wickerson

September 13, 2023


#### Abstract

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

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


## Contents

1 Introduction ..... 2
2 Finite partial functions ..... 3
2.1 Difference ..... 3
2.2 Comprehension ..... 3
2.3 Domain ..... 4
2.4 Lookup ..... 4
3 General purpose definitions and lemmas ..... 5
3.1 Projection functions on triples ..... 5
4 Proof chains ..... 6
4.1 Projections ..... 6
4.2 Chain length ..... 7
4.3 Extracting triples from chains ..... 7
4.4 Evaluating a predicate on each triple of a chain ..... 8
4.5 A map function for proof chains ..... 8
4.6 Extending a chain on its right-hand side ..... 8
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. ..... 14
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 quantifierfree 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: $A|\cap| B=\{| |\} \longleftrightarrow(\forall x . x|\in| A \longrightarrow x|\notin| B)$ $\langle p r o o f\rangle$
unbundle lifting-syntax
unbundle fmap.lifting
type-notation fmap (infix $\left.\rightharpoonup_{f} 9\right)$

### 2.1 Difference

## definition

```
    map-diff \(::\left({ }^{\prime} k \rightharpoonup{ }^{\prime} v\right) \Rightarrow{ }^{\prime} k\) fset \(\Rightarrow\left({ }^{\prime} k{ }^{\prime} v\right)\)
where
    map-diff \(f k s=\) restrict-map \(f(-f s e t k s)\)
```


## lift-definition

fmap-diff $::\left({ }^{\prime} k \rightharpoonup_{f}{ }^{\prime} v\right) \Rightarrow{ }^{\prime} k$ fset $\Rightarrow\left({ }^{\prime} k \rightharpoonup_{f}{ }^{\prime} v\right)($ infix $\ominus 110)$
is map-diff
$\langle p r o o f\rangle$

### 2.2 Comprehension

## definition

make-map :: ${ }^{\prime} k$ fset $\Rightarrow{ }^{\prime} v \Rightarrow\left({ }^{\prime} k{ }^{\prime} v\right)$
where
make-map $k s v \equiv \lambda k$. if $k \in f$ set $k s$ 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 $k s$
$\langle p r o o f\rangle$

## lift-definition

make-fmap $::{ }^{\prime} k$ fset $\Rightarrow{ }^{\prime} v \Rightarrow\left({ }^{\prime} k \rightharpoonup_{f}{ }^{\prime} v\right)([-\mid=>-])$
is make-map parametric make-map-transfer
$\langle$ proof $\rangle$
lemma make-fmap-empty $[$ simp $]:[\{| |\} \mid=>f]=$ fmempty $\langle p r o o f\rangle$

### 2.3 Domain

lemma fmap-add-commute:
assumes fmdom $A|\cap|$ fmdom $B=\{\|\}$
shows $A++_{f} B=B++_{f} A$
$\langle p r o o f\rangle$ including fset.lifting
$\langle p r o o f\rangle$
lemma make-fmap-union:

$$
[x s \mid=>v]++_{f}[y s \mid=>v]=[x s|\cup| y s \mid=>v]
$$

$\langle p r o o f\rangle$
lemma fdom-make-fmap: fmdom $[k s \mid=>v]=k s$

## $\langle p r o o f\rangle$

### 2.4 Lookup

## lift-definition

lookup :: $\left({ }^{\prime} k \rightharpoonup_{f}{ }^{\prime} v\right) \Rightarrow{ }^{\prime} k \Rightarrow{ }^{\prime} v$
is (o) the $\langle$ proof $\rangle$
lemma lookup-make-fmap:
assumes $k \in f$ set $k s$
shows lookup [ks $\mid=>v] k=v$
$\langle p r o o f\rangle$
lemma lookup-make-fmap1:
lookup $[\{|k|\} \mid=>v] k=v$
$\langle p r o o f\rangle$
lemma lookup-union1:
assumes $k|\in|$ fmdom ys
shows lookup (xs $+_{+_{f}}$ ys) $k=$ lookup ys $k$

```
<proof\rangle including fset.lifting
<proof\rangle
lemma lookup-union2:
    assumes k|\not\in| fmdom ys
    shows lookup (xs ++\mp@subsup{}{f}{}ys) k= lookup xs }
<proof\rangle including fset.lifting
<proof\rangle
lemma lookup-union3:
    assumes k| || fmdom xs
    shows lookup (xs ++\mp@subsup{+}{f}{}}\mathrm{ ys) k= lookup ys }
<proof\rangle including fset.lifting
<proof\rangle
end
```


## 3 General purpose definitions and lemmas

```
theory JHelper imports
```

    Main
    begin
lemma Collect-iff:
$a \in\{x . P x\} \equiv P a$
$\langle p r o o f\rangle$
lemma diff-diff-eq:
assumes $C \subseteq B$
shows $(A-C)-(B-C)=A-B$
$\langle p r o o f\rangle$
lemma nth-in-set:
$\llbracket i<$ length $x s ; x s!i=x \rrbracket \Longrightarrow x \in$ set $x s\langle p r o o f\rangle$
lemma disjI [intro]:
assumes $\neg P \Longrightarrow Q$
shows $P \vee Q$
$\langle p r o o f\rangle$
lemma empty-eq-Plus-conv:
$(\}=A<+>B)=(A=\{ \} \wedge B=\{ \})$
$\langle p r o o f\rangle$

### 3.1 Projection functions on triples

definition $f s t 3::{ }^{\prime} a \times{ }^{\prime} b{ }^{\prime} c \Rightarrow{ }^{\prime} a$
where $f_{s t}$. $\equiv f s t$

```
definition snd3 :: ' \(a \times 1 b \times{ }^{\prime} c \Rightarrow{ }^{\prime} b\)
where \(s n d 3 \equiv f s t \circ s n d\)
definition thd3 :: ' \(a \times\) ' \(b \times{ }^{\prime} c \Rightarrow{ }^{\prime} c\)
where \(t h d 3 \equiv\) snd \(\circ\) snd
lemma fst3-simp:
    \(\bigwedge a b c . f s t 3(a, b, c)=a\)
\(\langle p r o o f\rangle\)
lemma snd3-simp:
    \(\bigwedge a b c . \operatorname{snd} 3(a, b, c)=b\)
\(\langle p r o o f\rangle\)
lemma thd3-simp:
    \(\bigwedge a b c . \operatorname{thd} 3(a, b, c)=c\)
\(\langle p r o o f\rangle\)
lemma tripleI:
    fixes \(T U\)
    assumes fst3 \(T=f s t 3 U\)
        and snd3 \(T=\) snd3 \(U\)
        and thd3 \(T=\) thd \(3 U\)
    shows \(T=U\)
\(\langle\) proof \(\rangle\)
end
```


## 4 Proof chains

theory Proofchain imports

> JHelper
begin
An ( ${ }^{\prime} a,{ }^{\prime} c$ ) chain is a sequence of alternating ' $a$ 's and ${ }^{\prime} c$ 's, beginning and ending with an ' $a$. Usually ' $a$ represents some sort of assertion, and ${ }^{\prime} c$ 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 ({ - } \cdots - [ [0,0,0] 60)
```

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

### 4.1 Projections

Project first assertion.

```
fun
    pre : : ( \({ }^{\prime} a,{ }^{\prime} c\) ) chain \(\Rightarrow{ }^{\prime} a\)
where
    pre \(\{P\}=P\)
| pre \((\{P\} \cdots-\cdot)=P\)
```

Project final assertion.
fun
post $::\left({ }^{\prime} a,{ }^{\prime} c\right)$ chain $\Rightarrow{ }^{\prime} a$
where
post $\{P\}=P$
$\mid \operatorname{post}(\{-\} \cdot-\Pi)=$ post $\Pi$

Project list of commands.

## fun

comlist $::\left({ }^{\prime} a,{ }^{\prime} c\right)$ chain $\Rightarrow{ }^{\prime} c$ list
where
comlist $\{-\}=[]$
$\mid \operatorname{comlist}(\{-\} \cdot x \cdot \Pi)=x \#(\operatorname{comlist} \Pi)$

### 4.2 Chain length

## fun

chainlen $::\left({ }^{\prime} a,{ }^{\prime} c\right)$ chain $\Rightarrow$ nat
where
chainlen $\{-\}=0$
| chainlen $(\{-\} \cdot-\Pi)=1+($ chainlen $\Pi)$
lemma len-comlist-chainlen:
length $($ comlist $\Pi)=$ chainlen $\Pi$〈proof〉

### 4.3 Extracting triples from chains

$n t h t r i p l e ~ \Pi n$ extracts the $n$th triple of $\Pi$, counting from 0 . The function is well-defined when $n<$ chainlen $\Pi$.

## fun

nthtriple :: ( $\left.{ }^{\prime} a,^{\prime} c\right)$ chain $\Rightarrow$ nat $\Rightarrow\left({ }^{\prime} a *{ }^{\prime} c *^{\prime} a\right)$
where
nthtriple $(\{\mid P\} \cdot x \cdot \Pi) 0=(P, x$, pre $\Pi)$
| nthtriple $(\{P\} \cdot x \cdot \Pi)($ Suc $n)=$ nthtriple $\Pi n$
The list of middle components of $\Pi$ 's triples is the list of $\Pi$ 's commands.
lemma snds-of-triples-form-comlist:
fixes $\Pi i$
shows $i<$ chainlen $\Pi \Longrightarrow$ snd3 $($ nthtriple $\Pi i)=($ comlist $\Pi)!i$
$\langle p r o o f\rangle$

### 4.4 Evaluating a predicate on each triple of a chain

chain-all $\varphi$ holds of $\Pi$ iff $\varphi$ holds for each of $\Pi$ 's individual triples.
fun
chain-all $::\left(\left({ }^{\prime} a \times{ }^{\prime} c \times{ }^{\prime} a\right) \Rightarrow\right.$ bool $) \Rightarrow\left({ }^{\prime} a,^{\prime} c\right)$ chain $\Rightarrow$ bool
where
chain-all $\varphi\{\sigma\}=$ True
$\mid$ chain-all $\varphi(\{\sigma\} \cdot x \cdot \Pi)=(\varphi(\sigma, x$, pre $\Pi) \wedge$ chain-all $\varphi \Pi)$
lemma chain-all-mono [mono]:
$x \leq y \Longrightarrow$ chain-all $x \leq$ chain-all $y$
$\langle$ proof $\rangle$
lemma chain-all-nthtriple:
$($ chain-all $\varphi \Pi)=(\forall i<$ chainlen $\Pi . \varphi($ nthtriple $\Pi i))$
$\langle p r o o f\rangle$

### 4.5 A map function for proof chains

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

```
fun
    chainmap \(::\left({ }^{\prime} a \Rightarrow{ }^{\prime} b\right) \Rightarrow\left({ }^{\prime} c \Rightarrow{ }^{\prime} d\right) \Rightarrow\left({ }^{\prime} a,^{\prime} c\right)\) chain \(\Rightarrow\left({ }^{\prime} b,{ }^{\prime} d\right)\) chain
where
    chainmap \(f g\{P\}=\{f P\}\)
\(\mid\) chainmap \(f g(\{P\} \cdot x \cdot \Pi)=\{f P\} \cdot g x \cdot\) chainmap \(f g \Pi\)
```

Mapping over a chain preserves its length.
lemma chainmap-preserves-length:
chainlen (chainmap $f g \Pi$ ) $=$ chainlen $\Pi$
$\langle p r o o f\rangle$
lemma pre-chainmap:
pre $($ chainmap $f g \Pi)=f($ pre $\Pi)$
$\langle p r o o f\rangle$
lemma post-chainmap:
post $($ chainmap f $g \Pi)=f($ post $\Pi)$
$\langle p r o o f\rangle$
lemma nthtriple-chainmap:
assumes $i<$ chainlen $\Pi$
shows nthtriple (chainmap $f g$ П) $i$
$=(\lambda t .(f(f$ st3 $t), g(\operatorname{snd} 3 t), f($ thd3 t) $))($ nthtriple $\Pi i)$
$\langle p r o o f\rangle$

### 4.6 Extending a chain on its right-hand side <br> fun

```
    cSnoc :: ('a,'c) chain = ' c = ' }a=>('a,'c) chain
where
```




```
lemma len-snoc:
    fixes \Pi x P
    shows chainlen (cSnoc \Pix P)=1+(chainlen \Pi)
\langleproof\rangle
lemma pre-snoc:
    pre (cSnoc \PixP) = pre \Pi
<proof\rangle
lemma post-snoc:
    post (cSnoc П x P) = P
<proof\rangle
lemma comlist-snoc:
    comlist (cSnoc \Pi x p)= comlist \Pi @ [x]
<proof\rangle
```

end

## 5 Assertions, commands, and separation logic proof rules

## theory Ribbons-Basic imports Main <br> 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 staroperator, and existentially-quantified logical variables.
typedecl assertion
axiomatization
Emp :: assertion
axiomatization

```
    Star :: assertion = assertion }=>\mathrm{ assertion (infixr ^ 55)
where
    star-comm: }p\starq=q\starp\mathrm{ and
    star-assoc: }(p\starq)\starr=p\star(q\starr)\mathrm{ and
    star-emp: p\star Emp = p and
    emp-star: Emp \star p = p
lemma star-rot:
    q\star p\starr=p\star q\starr
\langleproof\rangle
axiomatization
    Exists :: string => assertion }=>\mathrm{ assertion
```

Extracting the set of program variables mentioned in an assertion.

```
axiomatization
    rd-ass :: assertion }=>\mathrm{ string set
where rd-emp:rd-ass Emp = {}
    and rd-star:rd-ass ( }p\starq)=rd\mathrm{ -ass }p\cuprd\mathrm{ -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 $\Rightarrow$ command $\Rightarrow$ command

## axiomatization

```
    Loop :: command \(\Rightarrow\) command
```


## axiomatization

Skip :: command

## axiomatization

Seq :: command $\Rightarrow$ command $\Rightarrow$ command (infixr ;; 55)
where seq-assoc: c1 $; ;(c 2 ; ; c 3)=(c 1 ; ; c 2) ; ; c 3$
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 $\Rightarrow$ string set
where rd-com-choose: $r d$-com (Choose c1 c2) $=r d$-com $c 1 \cup r d$-com $c 2$ and rd-com-loop: rd-com (Loop c) $=r d$-com $c$
and rd-com-skip: rd-com (Skip) $=\{ \}$
and $r d$－com－seq：$r d$－com $(c 1 ; ; c 2)=r d$－com $c 1 \cup r d$－com $c 2$
Extracting the set of program variables written by a command．

```
axiomatization
    wr-com :: command }=>\mathrm{ string set
where wr-com-choose: wr-com (Choose c1 c2) = wr-com c1 U 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 ;;c\mathcal{L})=wr-com c1 U 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 \(x q\) )
\(\mid\) choose: 【 prov-triple \((p, c 1, q) ;\) prov-triple \((p, c 2, q) \rrbracket\)
    \(\Longrightarrow\) prov-triple ( \(p\), Choose c1 c2, \(q\) )
| loop: prov-triple \((p, c, p) \Longrightarrow\) prov-triple ( \(p\), Loop \(c, p\) )
| frame: 【prov-triple \((p, c, q) ; w r-c o m(c) \cap r d-a s s(r)=\{ \} \rrbracket\)
    \(\Longrightarrow\) prov-triple \((p \star r, c, q \star r)\)
| skip: prov-triple ( \(p\), Skip, p)
| seq: 【 prov-triple \((p, c 1, q) ;\) prov-triple \((q, c 2, r) \rrbracket\)
    \(\Longrightarrow\) prov-triple \((p, c 1 ;<2, r)\)
```

Here are some derived proof rules，which are used in our ribbon－checking tool．
lemma choice－lemma：
assumes prov-triple $(p 1, c 1, q 1)$ and prov-triple $(p 2, c 2, q 2)$
and $p=p 1$ and $p 1=p 2$ and $q=q 1$ and $q 1=q 2$
shows prov-triple ( $p$, Choose c1 c2, $q$ )
$\langle p r o o f\rangle$
lemma loop-lemma:
assumes prov-triple ( $p 1, c, q 1$ ) and $p=p 1$ and $p 1=q 1$ and $q 1=q$
shows prov-triple ( $p$, Loop $c, q$ )
$\langle p r o o f\rangle$
lemma seq-lemma:
assumes prov-triple ( $p 1, c 1, q 1$ ) and prov-triple ( $p 2$, c2, q2)
and $q 1=p 2$
shows prov-triple ( $p 1, c 1 ; ; c 2, q 2$ )
$\langle p r o o f\rangle$
end

## 6 Ribbon proof interfaces

theory Ribbons-Interfaces imports<br>Ribbons-Basic<br>Proofchain<br>HOL-Library.FSet<br>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 ( $\varepsilon_{c}$ )
| Exists-int-conc string conc-interface
We define an equivalence on interfaces. The first three rules make this an equivalence relation. The next three make it a congruence. The next two identify interfaces up to associativity and commutativity of $\left(\otimes_{c}\right)$ The final two make $\varepsilon_{c}$ the left and right unit of $\left(\otimes_{c}\right)$.

```
inductive
    equiv-int \(::\) conc-interface \(\Rightarrow\) conc-interface \(\Rightarrow\) bool (infix \(\simeq 45\) )
where
    refl: \(P \simeq P\)
    sym: \(P \simeq Q \Longrightarrow Q \simeq P\)
|trans: \(\llbracket P \simeq Q ; Q \simeq R \rrbracket \Longrightarrow P \simeq R\)
| cong-hcomp1: \(P \simeq Q \Longrightarrow P^{\prime} \otimes_{c} P \simeq P^{\prime} \otimes_{c} Q\)
cong-hcomp2: \(P \simeq Q \Longrightarrow P \otimes_{c} P^{\prime} \simeq Q \otimes_{c} P^{\prime}\)
cong-exists: \(P \simeq Q \Longrightarrow\) Exists-int-conc \(x P \simeq\) Exists-int-conc x \(Q\)
hcomp-conc-assoc: \(P \otimes_{c}\left(Q \otimes_{c} R\right) \simeq\left(P \otimes_{c} Q\right) \otimes_{c} R\)
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:
    \(\llbracket P \simeq Q ; P^{\prime} \simeq Q^{\prime} \rrbracket \Longrightarrow P \otimes_{c} P^{\prime} \simeq Q \otimes_{c} Q^{\prime}\)
\(\langle p r o o f\rangle\)
```

quotient-type interface $=$ conc-interface $/$ equiv-int
$\langle p r o o f\rangle$

```
lift-definition
    Ribbon :: assertion => interface
is Ribbon-conc \langleproof\rangle
```


## lift-definition

```
    Emp-int :: interface (\varepsilon)
is }\mp@subsup{\varepsilon}{c}{}\langleproof
```


## lift-definition

```
                            Exists-int :: string }=>\mathrm{ interface }=>\mathrm{ interface
is Exists-int-conc
<proof\rangle
lift-definition
    HComp-int :: interface }=>\mathrm{ interface }=>\mathrm{ interface (infix & 50)
is HComp-int-conc 〈proof\rangle
lemma hcomp-comm:
    (P\otimesQ)=(Q\otimesP)
\langleproof\rangle
lemma hcomp-assoc:
    (P\otimes (Q\otimesR)) = ((P\otimesQ)\otimesR)
<proof\rangle
lemma emp-hcomp:
    \varepsilon\otimesP=P
<proof\rangle
lemma hcomp-emp:
    P\otimes\varepsilon=P
<proof\rangle
lemma comp-fun-commute-hcomp:
    comp-fun-commute ( }\otimes\mathrm{ )
\langleproof\rangle
```

```
6.2 An iterated horizontal-composition operator
definition iter-hcomp :: ('a fset) \(\Rightarrow\) ('a \(\Rightarrow\) interface) \(\Rightarrow\) interface
where
    iter-hcomp \(X f \equiv\) ffold \(((\otimes) \circ f) \varepsilon X\)
syntax iter-hcomp-syntax ::
    \({ }^{\prime} a \Rightarrow\left({ }^{\prime} a\right.\) fset \() \Rightarrow\left({ }^{\prime} a \Rightarrow\right.\) interface \() \Rightarrow\) interface
            \(((\otimes-|\in|-.-)[0,0,10] 10)\)
translations \(\bigotimes x|\in| M . e==C O N S T\) iter-hcomp \(M(\lambda x . e)\)
```

term $\otimes P|\in| P s . f P$－this is eta－expanded，so prints in expanded form term $\otimes P|\in| P s . f$－this isn＇t eta－expanded，so prints as written
lemma iter－hcomp－cong：
assumes $\forall v \in$ fset vs．$\varphi v=\varphi^{\prime} v$
shows $(\otimes v|\in| v s . \varphi v)=\left(\otimes v|\in| v s . \varphi^{\prime} v\right)$
〈proof〉
lemma iter－hcomp－empty：
shows $(\otimes x|\in|\{|\mid\} . p x)=\varepsilon$
〈proof〉
lemma iter－hcomp－insert：
assumes $v|\notin|$ ws
shows $(\otimes x|\in|$ finsert $v$ ws．$p x)=(p v \otimes(\otimes x|\in|$ ws．$p x))$
〈proof〉
lemma iter－hcomp－union：
assumes $v s|\cap|$ ws $=\{| |\}$
shows $(\otimes x|\in|$ vs $|\cup|$ ws．$p x)=((\bigotimes x|\epsilon|$ vs．$p x) \otimes(\bigotimes x|\in|$ ws．$p x))$
$\langle$ proof $\rangle$

## 6．3 Semantics of interfaces

The semantics of an interface is an assertion．

```
fun
    conc-asn :: conc-interface }=>\mathrm{ assertion
where
    conc-asn (Ribbon-conc p)=p
| conc-asn (P 卶 Q ) = (conc-asn P)\star (conc-asn Q)
| conc-asn ( }\mp@subsup{\varepsilon}{c}{})=Em
| conc-asn (Exists-int-conc x P) = Exists x (conc-asn P)
```


## lift－definition

```
    asn :: interface }=>\mathrm{ assertion
```

is conc-asn
$\langle p r o o f\rangle$
lemma asn-simps [simp]:
asn $($ Ribbon $p)=p$
$\operatorname{asn}(P \otimes Q)=($ asn $P) \star(\operatorname{asn} Q)$
asn $\varepsilon=E m p$
asn $($ Exists-int $x P)=$ Exists $x($ asn $P)$
$\langle p r o o f\rangle$

## 6．4 Program variables mentioned in an interface． fun

```
    rd-conc-int :: conc-interface => string set
where
    rd-conc-int (Ribbon-conc p) =rd-ass p
|rd-conc-int (P \otimes c Q ) = rd-conc-int P }\cup\mathrm{ rd-conc-int Q
|rd-conc-int ( }\mp@subsup{\varepsilon}{c}{})={
| rd-conc-int (Exists-int-conc x P)}=rd-conc-int P
```


## lift－definition

rd－int $::$ interface $\Rightarrow$ string set
is $r d$－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:
    \. \llbracketlength cs = chainlen \Pi ; p1 = asn (pre \Pi) ; p2 = asn (post \Pi);
    \i.i<chainlen \Pi \Longrightarrow prov-triple
    (asn (fst3 (nthtriple \Pi i)), cs ! i, asn (thd3 (nthtriple \Pi i)))】
    "prov-triple (p1, foldr (;;) cs Skip, p2)
<proof\rangle
end
```


## 7 Syntax and proof rules for stratified diagrams

## theory Ribbons－Stratified imports <br> Ribbons－Interfaces <br> Proofchain <br> begin

We define the syntax of stratified diagrams．We give proof rules for strati－ fied 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 \(\times\) interface
Extracting the command from a stratified diagram.
fun
    com-sdia \(::\) sdiagram \(\Rightarrow\) command and
    com-cell :: cell \(\Rightarrow\) command
where
    com-sdia \((\) SDiagram @s) \(=\) foldr \((; ;)(\) map \((c o m-c e l l \circ f s t) \varrho s)\) Skip
| com-cell \((\) Filler P) \(=\) Skip
| com-cell (Basic P c Q) \(=c\)
| com-cell (Exists-sdia x D) \(=\) com-sdia \(D\)
| com-cell (Choose-sdia P D E Q) \(=\) Choose (com-sdia D) (com-sdia E)
| com-cell (Loop-sdia P D \(Q\) ) \(=\) Loop (com-sdia \(D)\)
Extracting the program variables written by a stratified diagram.
```


## fun

```
    wr-sdia :: sdiagram \(\Rightarrow\) string set and
```

    wr-sdia :: sdiagram \(\Rightarrow\) string set and
    wr-cell :: cell \(\Rightarrow\) string set
    where
wr-sdia $(S D i a g r a m ~ \varrho s)=(\bigcup r \in$ set $\varrho s . w r$-cell $(f s t r))$
$\mid$ 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 $D \cup$ 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 \varrhos :: row list
and \varrho :: row
shows(wr-sdia D = wr-com (com-sdia D))
and (wr-cell \gamma = wr-com (com-cell \gamma))
and (U\varrho\in set \varrhos.wr-cell (fst \varrho))
=wr-com (foldr (;;) (map (\lambda(\gamma,F). com-cell \gamma) \varrhos) Skip)
and wr-cell (fst \varrho) = wr-com(com-cell (fst \varrho))
<proof>

```

\subsection*{7.2 Proof rules for stratified diagrams}

\section*{inductive}
prov-sdia :: [sdiagram, interface, interface] \(\Rightarrow\) bool and
prov-row :: [row, interface, interface] \(\Rightarrow\) bool and
prov-cell \(::[\) cell, interface, interface \(] \Rightarrow\) bool

\section*{where}

SRibbon: prov-cell (Filler P) P P
```

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

```

\subsection*{7.3 Soundness}
lemma soundness-strat-helper:
```

    (prov-sdia D P Q \longrightarrow prov-triple (asn P, com-sdia D, asn Q)) ^
    (prov-row \varrho P Q \longrightarrow prov-triple (asn P, com-cell (fst \varrho), asn Q)) ^
    (prov-cell \gamma P Q 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\rangle
end

```

\section*{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.

\subsection*{8.1 Syntax of graphical diagrams}

Fix a type for node identifiers
typedecl node
Note that this datatype is necessarily an overapproximation of syntacticallywellformed 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 $f$ set $\times$ command-gadget $\times$ node fset
Projecting components from a graph
fun vertices $::$ diagram $\Rightarrow$ node fset $(-\wedge V[1000] 1000)$
where (Graph $V \Lambda E)^{\wedge} V=V$
term this $\left(\right.$ is $\left.^{\wedge} V\right)=(\text { a test })^{\wedge} V$
fun labelling :: diagram $\Rightarrow$ labelling $(-\wedge \Lambda$ [1000] 1000)
where $(G r a p h V \Lambda E) \wedge \Lambda=\Lambda$
fun edges $::$ diagram $\Rightarrow$ edge list $(-\wedge E[1000] 1000)$
where $(\text { Graph } V \Lambda E)^{\wedge} E=E$

```

\subsection*{8.2 Well formedness of graphical diagrams}
```

definition acyclicity :: edge list $\Rightarrow$ bool where
acyclicity $E \equiv$ acyclic $(\bigcup e \in \operatorname{set} E . f s e t(f s t 3 e) \times f s e t(t h d 3 e))$
definition linearity $::$ edge list $\Rightarrow$ bool
where
linearity $E \equiv$
distinct $E \wedge(\forall e \in \operatorname{set} E . \forall f \in$ set $E . e \neq f \longrightarrow$
fst3 e $|\cap|$ fst3 $f=\{\|\} \wedge$
thd3 $e|\cap|$ thd3 $f=\{| |\}$ )
lemma linearity D:
assumes linearity $E$
shows distinct $E$
and $\bigwedge e f . \llbracket e \in \operatorname{set} E ; f \in \operatorname{set} E ; e \neq f \rrbracket \Longrightarrow$
fst3 $e|\cap| f s t 3 f=\{| |\} \wedge$
thd3 e $|\cap| \operatorname{thd} 3 f=\{| |\}$
$\langle p r o o f\rangle$
lemma linearityD2:
linearity $E \Longrightarrow(\forall e f . e \in$ set $E \wedge f \in \operatorname{set} E \wedge e \neq f \longrightarrow$
fst3 $e|\cap|$ fst3 $f=\{\| \mid\} \wedge$
thd3 $e|\cap|$ thd3 $f=\{| |\})$
$\langle p r o o f\rangle$

```

\section*{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 \(x\) )
| wf-com: wf-com (Com c)
|wf-choice: \(\llbracket w f\)-dia \(G\); wf-dia H \(\rrbracket \Longrightarrow w f\)-com (Choose-dia G H)
|wf-loop: wf-dia \(G \Longrightarrow w f\)-com (Loop-dia \(G\) )
| wf-dia: \(\llbracket \forall e \in\) set \(E\). wf-com (snd3 e) ; \(\forall v \in f\) set \(V\).wf-ass ( \(\Lambda v\) ); acyclicity \(E ;\) linearity \(E ; \forall e \in\) set \(E . f s t 3\) 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 \(E . f s t 3\) e \(|\cup|\) thd3 \(e|\subseteq| V\)
\(\langle p r o o f\rangle\)

\subsection*{8.3 Initial and terminal nodes}

\section*{definition}
initials \(::\) diagram \(\Rightarrow\) node fset
where
\[
\text { initials } G=\text { ffilter }\left(\lambda v .\left(\forall e \in \text { set } G^{\wedge} E . v|\notin| \text { thd3 e)) } G^{\wedge} V\right.\right.
\]

\section*{definition}
terminals \(:\) diagram \(\Rightarrow\) node fset
where
terminals \(G=\) ffilter \(\left(\lambda v .\left(\forall e \in\right.\right.\) set \(\left.\left.G^{\wedge} E . v|\notin| f s t 3 e\right)\right) G^{\wedge} V\)
lemma no-edges-imp-all-nodes-initial: initials \((\) Graph \(V \Lambda[])=V\)
\(\langle p r o o f\rangle\)
lemma no-edges-imp-all-nodes-terminal: terminals \((\) Graph \(V \Lambda[])=V\)
\(\langle\) proof〉
lemma initials-in-vertices:
initials \(G|\subseteq| G^{\wedge} V\)
\(\langle\) proof \(\rangle\)
```

lemma terminals-in-vertices:
terminals G |\subseteq| G`V
\langleproof\rangle

```

\subsection*{8.4 Top and bottom interfaces}

\section*{primrec}
```

    top-ass :: assertion-gadget \(\Rightarrow\) interface and
    ```
    top-dia :: diagram \(\Rightarrow\) interface
where
    top-dia \((\) Graph \(V \Lambda E)=(\bigotimes v|\in|\) initials \((G r a p h ~ V \Lambda E)\). top-ass \((\Lambda v))\)
|top-ass (Rib p) \(=\) Ribbon \(p\)
| top-ass (Exists-dia x \(G\) ) = Exists-int \(x(\) top-dia \(G)\)

\section*{primrec}
    bot-ass :: assertion-gadget \(\Rightarrow\) interface and
    bot-dia :: diagram \(\Rightarrow\) interface
where
    bot-dia \((\) Graph \(V \Lambda E)=(\otimes v|\in|\) terminals \((G r a p h ~ V \Lambda E)\).bot-ass \((\Lambda v))\)
    bot-ass \((\) Rib \(p)=\) Ribbon \(p\)
bot-ass \((\) Exists-dia \(x G)=\) Exists-int \(x(\) bot-dia \(G)\)

\subsection*{8.5 Proof rules for graphical diagrams}

\section*{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 \((\operatorname{Com} c) P Q\)
Choice: 【prov-dia G P \(Q\); prov-dia H P Q 】
\(\Longrightarrow\) prov-com (Choose-dia G H) PQ
| Loop: prov-dia \(G P P \Longrightarrow\) prov-com (Loop-dia G) \(P\) P
\(\mid\) Main: \(\llbracket w f\)-dia \(G ; \Lambda v . v \in f\) set \(G^{\wedge} V \Longrightarrow\) prov-ass \(\left(G^{\wedge} \Lambda v\right)\);
\(\bigwedge e . e \in \operatorname{set} G^{\wedge} E \Longrightarrow\) prov-com (snd3 e)
\(\left(\otimes v|\in|\right.\) fst3 e. bot-ass \(\left.\left(G^{\wedge} \Lambda v\right)\right)\)
\(\left(\otimes v|\in|\right.\) thd3 e. top-ass \(\left.\left(G^{\wedge} \Lambda v\right)\right) \rrbracket\)
\(\Longrightarrow\) prov-dia \(G(\) top-dia \(G)(\) 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) PQ
inductive-cases exists-inv: prov-ass (Exists-dia \(x G\) )
inductive-cases skip-inv: prov-ass (Rib p)

\subsection*{8.6 Extracting commands from diagrams}
```

type-synonym lin = (node + edge) list

```

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

definition lins $::$ diagram $\Rightarrow$ lin set
where
lins $G \equiv\{\pi::$ lin.
( distinct $\pi$ )
$\wedge\left(\right.$ set $\pi=\left(f\right.$ set $\left.G^{\wedge} V\right)<+>\left(\right.$ set $\left.\left.G^{\wedge} E\right)\right)$
$\wedge(\forall i j v e . i<$ 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 $\pi$ )
and $\left(\right.$ set $\pi=\left(\right.$ fset $\left.G^{\wedge} V\right)<+>\left(\right.$ set $\left.\left.G^{\wedge} E\right)\right)$
and $(\forall i j v e . i<$ length $\pi \wedge j<$ length $\pi$
$\wedge \pi!i=\operatorname{Inl} v \wedge \pi!j=\operatorname{Inr} e \wedge v|\in| f s t 3 e \longrightarrow i<j)$
and $(\forall j k w e . j<$ length $\pi \wedge k<$ length $\pi$
$\wedge \pi!j=$ Inr $e \wedge \pi!k=\operatorname{Inl} w \wedge w|\in|$ thd $33 \longrightarrow j<k)$
$\langle p r o o f\rangle$

```

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 $)(\pi!i)(c s!i))$
$=$
$((\forall i . i<$ length $\pi \wedge(\exists v .(\pi!i)=$ Inl $v) \longrightarrow$
coms-ass $(\Lambda(\operatorname{projl}(\pi!i)))(c s!i)) \wedge$
$(\forall i . i<$ length $\pi \wedge(\exists e .(\pi!i)=\operatorname{Inr} e) \longrightarrow$
coms-com $($ snd3 $(\operatorname{projr}(\pi!i)))(c s!i)))$
$\langle p r o o f\rangle$

```

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 \(\pi\) of these nodes and edges (using lins), and composing the extracted commands in accordance with \(\pi\).
```

inductive
coms-dia :: [diagram, command] }=>\mathrm{ bool and

```
```

    coms-ass :: [assertion-gadget, command] }=>\mathrm{ bool and
    coms-com :: [command-gadget, command] }=>\mathrm{ bool
    where
coms-skip: coms-ass (Rib p) Skip
| coms-exists:coms-dia G c \Longrightarrow coms-ass (Exists-dia x G)c
| coms-basic: coms-com (Com c) c
| coms-choice: \llbracket coms-dia G c; coms-dia H d\rrbracket\Longrightarrow
coms-com (Choose-dia G H) (Choose c d)
| coms-loop:coms-dia G c\Longrightarrow coms-com (Loop-dia G) (Loop c)
| coms-main: \llbracket \pi \in lins (Graph V \Lambda E); length cs = length \pi;
\foralli<length \pi.case-sum (coms-ass ○ \Lambda) (coms-com ○ snd3) (\pi!i) (cs!i)\rrbracket
"coms-dia (Graph V \Lambda 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')}
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 Gc
end

```

\section*{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.

\subsection*{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 \(\mid\) 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 \(p s\)-chain \(=(\) proofstate, node + edge \()\) chain
The next-ps \(\sigma\) function processes one node or one edge in a diagram, given the current proofstate \(\sigma\). 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 \(v s\) and \(w s\) respectively) by removing all the mappings from vs to Bot, and adding mappings from ws to Top.
```

fun next-ps :: proofstate $\Rightarrow$ node + edge $\Rightarrow$ proofstate
where
next-ps $\sigma($ Inl $v)=\sigma \ominus\{|v|\}++_{f}[\{|v|\} \mid=>$ Bot $]$
$\mid$ next-ps $\sigma($ Inr $e)=\sigma \ominus f$ ft3 $e++_{f}[$ thd 3 e $\mid=>$ Top $]$

```

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

\section*{definition}
\(m k\)-ps-chain \(::[\) ps-chain, \((\) node + edge \()\) list \(] \Rightarrow\) ps-chain
where
\(m k-p s-c h a i n \equiv\) foldl \((\lambda \Pi x . c S n o c ~ \Pi x(n e x t-p s(\) post \(\Pi) x))\)
lemma mk-ps-chain-preserves-length:
fixes \(\pi \Pi\)
shows chainlen ( \(m k\)-ps-chain \(\Pi \pi\) ) \(=\) chainlen \(\Pi+\) length \(\pi\)
\(\langle p r o o f\rangle\)
Distributing \(m k\)-ps-chain over (\#).
lemma mk-ps-chain-cons:
\(m k\)-ps-chain \(\Pi(x \# \pi)=m k\)-ps-chain \((c S n o c ~ \Pi x(n e x t-p s(\) post \(\Pi) x)) \pi\) \(\langle p r o o f\rangle\)

Distributing \(m k\)-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\rangle
Distributing mk-ps-chain over cCons.
lemma mk-ps-chain-ccons:
fixes }\pi

```

```

<proof\rangle

```
```

lemma pre-mk-ps-chain:

```
lemma pre-mk-ps-chain:
    fixes \Pi}
    fixes \Pi}
    shows pre (mk-ps-chain \Pi\pi) = pre \Pi
    shows pre (mk-ps-chain \Pi\pi) = pre \Pi
<proof\rangle
```

<proof\rangle

```

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

lemma comlist-mk-ps-chain:
comlist (mk-ps-chain \Pi\pi) = comlist \Pi @ }
<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 lins2 :: [node fset, diagram $] \Rightarrow$ lin set
where
lins2 $S G \equiv\{\pi::$ lin.
(distinct $\pi$ )
$\wedge\left(\right.$ set $\pi=\left(f\right.$ set $G^{\wedge} V-f$ set $\left.\left.S\right)<+>\operatorname{set} G^{\wedge} E\right)$
$\wedge(\forall i j v e . i<$ 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 $\pi$
$\wedge \pi!j=\operatorname{Inr} e \wedge \pi!k=\operatorname{Inl} w \wedge w|\in|$ thd3 $e \longrightarrow j<k)\}$

```
```

lemma lins2D:
assumes \pi}\inlins2 S
shows distinct }
and set \pi}=(f\mathrm{ set }\mp@subsup{G}{}{\wedge}V-fset S)<+> set G^
and <br>ijve.\llbracketi< length \pi; j< length \pi;
\pi!i=Inl v;\pi!j= Inr e;v|||fst3 e\rrbracket\Longrightarrow \<j
and }\bigwedgeikwe.\llbracketj<length \pi;k< length \pi
\pi!j= Inr e ; \pi!k= Inl w;w |\in| thd3 e\rrbracket\Longrightarrowj<k
\langleproof\rangle
lemma lins2I:
assumes distinct \pi
and set \pi}=(f\mathrm{ set G}\mp@subsup{G}{}{\wedge}V-fset S)<+> set G^
and <br>ijve.\llbracketi< length \pi;j< length \pi;
\pi!i=Inlv;\pi!j= Inr e; v |\in| fst3 e\rrbracket\Longrightarrow \Longrightarrow<j
and }\bigwedgejkwe.\llbracketj< length \pi;k< length \pi
\pi!j= Inr e;\pi!k= Inl w;w||| thd3 e\rrbracket\Longrightarrowj<k
shows }\pi\in\mathrm{ lins2 S G
<proof\rangle

```

When \(S\) is empty, the two definitions coincide.
lemma lins-is-lins2-with-empty-S:
lins \(G=\operatorname{lins2}\{\|\} G\)
\(\langle p r o o f\rangle\)
The first proofstate for a diagram \(G\) is obtained by mapping each of its initial nodes to Top.
```

definition
initial-ps $::$ diagram $\Rightarrow$ proofstate
where
initial-ps $G \equiv[$ initials $G \mid=>$ Top $]$

```

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

definition
initial-ps2 :: [node fset, diagram] }=>\mathrm{ proofstate
where
initial-ps2 S G \equiv[ initials G-S => Top] ++\mp@subsup{+}{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 \(\{\|\}\)
\(\langle p r o o f\rangle\)
The following function extracts the set of proofstate chains from a diagram.

\section*{definition}
ps-chains :: diagram \(\Rightarrow\) ps-chain set

\section*{where}
\[
\text { ps-chains } G \equiv m k \text {-ps-chain }(c N i l(\text { initial-ps } G)) ‘ \text { lins } G
\]

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

definition
ps-chains2 :: [node fset, diagram] }=>\mathrm{ ps-chain set
where
ps-chains2 S G \equivmk-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 {|}

```
\(\langle p r o o f\rangle\)
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 \(\left(\sigma++_{f}\right.\) disjoint \([\{|v|\} \mid=>\) Top \(]\), Inl \(v, \sigma++_{f} \operatorname{disjoint}[\{|v|\} \mid=>\) Bot \(])\), where \(v\) is a node, or of the form \(\left(\sigma++_{f} \operatorname{disjoint}[\{|v s|\} \mid=>\operatorname{Bot}\right.\) ], Inr \(e, \sigma+_{f}\) disjoint \([\{|w s|\} \mid=>\) 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 $\times($ node + edge $) \times$ proofstate $\Rightarrow$ bool
where
wf-ps-triple $T=($ case snd3 $T$ of
Inl $v \Rightarrow(\exists \sigma . v|\notin|$ fmdom $\sigma$
$\wedge$ fst3 $T=[\{|v|\} \mid=>$ Top $]++_{f} \sigma$
$\wedge \operatorname{thd3} T=[\{|v|\} \mid=>$ Bot $\left.]++_{f} \sigma\right)$
| Inr $e \Rightarrow(\exists \sigma$. (fst3 $e|\cup|$ thd3 $e)|\cap|$ fmdom $\sigma=\{| |\}$
$\wedge$ fst3 $T=\left[\left.\begin{array}{l}\text { fst3 } \\ e\end{array} \right\rvert\,=>\right.$ Bot $]++_{f} \sigma$
$\wedge$ thd3 $T=[$ thd3 e $\mid=>$ Top $\left.\left.]++_{f} \sigma\right)\right)$
lemma wf-ps-triple-nodeI:
assumes $\exists \sigma . v|\notin|$ fmdom $\sigma \wedge$
$\sigma 1=[\{|v|\} \mid=>$ Top $]++_{f} \sigma \wedge$
$\sigma \mathcal{Z}=[\{|v|\} \mid=>$ Bot $]++_{f} \sigma$
shows wf-ps-triple ( $\sigma 1$, Inl $v, \sigma 2$ )
$\langle p r o o f\rangle$
lemma wf-ps-triple-edgeI:
assumes $\exists \sigma$. (fst3 e |ن| thd3 e) $|\cap|$ fmdom $\sigma=\{| |\}$
$\wedge \sigma 1=[$ fst3 e $\mid=>$ Bot $]++_{f} \sigma$
$\wedge \sigma 2=[$ thd3 e $\mid=>$ Top $]++_{f} \sigma$
shows wf-ps-triple ( $\sigma 1$, Inr e, $\sigma$ 2)

```
```

\langleproof\rangle
definition
wf-ps-chain :: ps-chain => bool
where
wf-ps-chain \equivchain-all wf-ps-triple
lemma next-initial-ps2-vertex:
initial-ps2 ({|v|}|\cup|S)G
= initial-ps2 S G\ominus {|v|} +++f}[{|v|}|=> Bot
<proof\rangle
lemma next-initial-ps2-edge:
assumes G = Graph V \Lambda E and G}\mp@subsup{G}{}{\prime}=Graph V'\ \Lambda E' and
V'}=V-fst3 e and E' = removeAll e E and e\in set E and
fst3 e |\subseteq\S and S |\subseteq| initials G and wf-dia G
shows initial-ps2 (S - fst3 e) G' =
initial-ps2 S G\ominus fst3 e ++ff [thd3 e |=> Top ]
<proof>
lemma next-lins2-vertex:
assumes Inl v \# \pi \in lins2 S G
assumes v||| S
shows }\pi\in\operatorname{lins2({|v|}|\cup|S)G
<proof\rangle
lemma next-lins2-edge:
assumes Inr e\# \# \in lins2 S (Graph V \Lambda E)
and vs }<br>subseteq\
and e=(vs,c,ws)
shows }\pi\inlins2(S-vs)(Graph (V-vs) \Lambda(removeAll e E))
<proof\rangle

```

We wish to prove that every proofstate chain that can be obtained from a linear extension of \(G\) is well-formed and has as its final proofstate that state in which every terminal node in \(G\) is mapped to Bot.
We first prove this for partially-processed diagrams, for then the result for ordinary diagrams follows as an easy corollary.
We use induction on the size of the partially-processed diagram. The size of a partially-processed diagram \((G, S)\) is defined as the number of nodes in \(G\), plus the number of edges, minus the number of nodes in \(S\).
```

lemma wf-chains2:
fixes }
assumes S |\subseteq\ initials G
and wf-dia G
and \Pi\inps-chains2 S G
and fcard G G}V+\mathrm{ length }\mp@subsup{G}{}{\wedge}E=k+fcard
shows wf-ps-chain \Pi ^(post \Pi=[ terminals G |=> Bot ])

```
```

\langleproof\rangle
corollary wf-chains:
assumes wf-dia G
assumes \Pi\in ps-chains G
shows wf-ps-chain \Pi ^ post \Pi=[ terminals G |=> Bot ]
\langleproof\rangle

```

\subsection*{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] }=>\mathrm{ interface
where
ps-to-int G\sigma\equiv
\otimesv| || fmdom \sigma. case-topbot top-ass bot-ass(lookup \sigma v)(G^\Lambda v)
definition
ps-chain-to-int-chain :: [diagram, ps-chain] }=>\mathrm{ int-chain
where
ps-chain-to-int-chain G \Pi\equiv
chainmap (ps-to-int G) ((case-sum (Inl \circ G^\Lambda) (Inr \circ snd3))) \Pi
lemma ps-chain-to-int-chain-simp:
ps-chain-to-int-chain (Graph V \Lambda E) \Pi=
chainmap (ps-to-int (Graph V \Lambda E)) ((case-sum (Inl \circ \Lambda) (Inr \circ snd3))) \Pi
<proof\rangle

```

\subsection*{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 sidecondition. One way to obtain a separation logic lacking a side-condition on its frame rule is to use variables-as- resource.
We proceed by induction on the proof rules for graphical diagrams. We show that: (1) if a diagram \(G\) is provable w.r.t. interfaces \(P\) and \(Q\), then \(P\) and \(Q\) are the top and bottom interfaces of \(G\), and that the Hoare triple (asn \(P, c\), asn \(Q\) ) is provable for each command \(c\) that can be extracted from \(G\); (2) if a command-gadget \(C\) is provable w.r.t. interfaces \(P\) and \(Q\), then the Hoare triple (asn \(P, c\), asn \(Q\) ) is provable for each command \(c\) that
can be extracted from \(C\); and (3) if an assertion-gadget \(A\) is provable, and if the top and bottom interfaces of \(A\) are \(P\) and \(Q\) respectively, then the Hoare triple (asn \(P, c\), asn \(Q\) ) is provable for each command \(c\) that can be extracted from \(A\).
```

lemma soundness-graphical-helper:
assumes no-var-interference: $\bigwedge c$. wr-com $c=\{ \}$
shows
(prov-dia G P $Q \longrightarrow$
( $P=$ top-dia $G \wedge Q=$ bot-dia $G \wedge$
$(\forall c$ coms-dia $G c \longrightarrow$ prov-triple $($ asn $P, c$, asn $Q)))$ )
$\wedge($ prov-com $C P Q \longrightarrow$
$(\forall c$. coms-com $C c \longrightarrow$ prov-triple $($ asn $P, c$, asn $Q)))$
$\wedge$ (prov-ass $A \longrightarrow$
$(\forall c . c o m s-a s s A c \longrightarrow$ prov-triple $($ asn $(t o p-a s s A), c$, asn $(b o t-a s s A))))$
$\langle p r o o f\rangle$

```

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 \c. wr-com c={}
assumes prov-dia G PQ
shows }\forallc.coms-dia Gc\longrightarrow prov-triple (asn P, c, asn Q
<proof\rangle
end

```

\section*{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.```

