Ribbon Proofs for Separation Logic  
(Isabelle Formalisation)

John Wickerson  
August 16, 2018

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 .... 7
  4.5 A map function for proof chains .......... 8
  4.6 Extending a chain on its right-hand side .... 8
1 Introduction

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

Compared to traditional ‘proof outlines’, ribbon proofs emphasise the structure of a proof, so are intelligible and pedagogical. Because they contain less redundancy than proof outlines, and allow each proof step to be checked locally, they may be more scalable. Where proof outlines are cumbersome to modify, ribbon proofs can be visually manoeuvred to yield proofs of variant programs.
In this document, we formalise a two-dimensional graphical syntax for ribbon proofs, provide proof rules, and show that any provable ribbon proof can be recreated using the ordinary rules of separation logic. In fact, we provide two different formalisations. Our “stratified” formalisation sees a ribbon proof as a sequence of rows, with each row containing one step of the proof. This formalisation is very simple, but it does not reflect the visual intuition of ribbon proofs, which suggests that some proof steps can be slid up or down without affecting the validity of the overall proof. Our “graphical” formalisation sees a ribbon proof as a graph; specifically, as a directed acyclic nested graph. Ribbon proofs formalised in this way are more manoeuvrable, but proving soundness is trickier, and requires the assumption that separation logic’s Frame rule has no side-condition (an assumption that can be validated by using, for instance, variables-as-resource [2]).

2 Finite partial functions

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

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

2.2 Comprehension

definition make-map :: 'k fset ⇒ 'v ⇒ ('k ⇒ 'v)
where
make-map ks v ≡ λk. if k ∈ fset ks then Some v else None

lemma make-map-transfer[transfer-rule]: (rel-fset (=) ===> A ===> rel-map A) make-map make-map
⟨proof⟩
lemma dom-make-map:
  \( \text{dom} \ (\text{make-map} \ k s v) = \text{fset} \ k s \) 
  \(\langle\text{proof}\rangle\)

lift-definition
  \(\text{make-fmap} :: 'k \text{fset} \Rightarrow 'v \Rightarrow ('k \rightarrow f 'v) \ (\ [ \Rightarrow \ ] \ )\) 
  is \(\text{make-map} \ \text{parametric} \ \text{make-map-transfer}\) 
  \(\langle\text{proof}\rangle\)

lemma make-fmap-empty[simp]: \(\{\}\Rightarrow f = \text{fmempty}\) 
  \(\langle\text{proof}\rangle\)

2.3 Domain

lemma fmap-add-commute:
  \(\text{assumes} \ \text{fdom} \ A \ |\cap| \ \text{fdom} \ B = \{\}\) 
  \(\text{shows} \ A ++_f B = B ++_f A\) 
  \(\langle\text{proof}\rangle \ \text{including} \ \text{fset.lifting}\) 
  \(\langle\text{proof}\rangle\)

lemma make-fmap-union:
  \(\ [ \ xs \Rightarrow v \ ] ++_f \ [ \ ys \Rightarrow v \ ] = \ [ \ xs \cup \ ys \Rightarrow v \ ]\) 
  \(\langle\text{proof}\rangle\)

lemma fdom-make-fmap: \(\text{fdom} \ [ \ ks \Rightarrow v \ ] = ks\) 
  \(\langle\text{proof}\rangle\)

2.4 Lookup

lift-definition
  \(\text{lookup} :: ('k \rightarrow f 'v) \Rightarrow 'k \Rightarrow 'v\) 
  is \((\odot)\) the \(\langle\text{proof}\rangle\)

lemma lookup-make-fmap:
  \(\text{assumes} \ k \in \text{fset} \ k s\) 
  \(\text{shows} \ \text{lookup} \ [ \ ks \Rightarrow v \ ] \ k = v\) 
  \(\langle\text{proof}\rangle\)

lemma lookup-make-fmap1:
  \(\text{lookup} \ [ \ \{k\} \Rightarrow v \ ] \ k = v\) 
  \(\langle\text{proof}\rangle\)

lemma lookup-union1:
  \(\text{assumes} \ k \in \ text{fdom} \ ys\) 
  \(\text{shows} \ \text{lookup} \ (xs ++_f ys) \ k = \text{lookup} \ ys \ k\) 
  \(\langle\text{proof}\rangle \ \text{including} \ \text{fset.lifting}\) 
  \(\langle\text{proof}\rangle\)
lemma lookup-union2:
assumes k \notin \text{fmdom } ys
shows lookup (xs ++ f \ ys) k = lookup xs k
⟨proof⟩ including fset.lifting
⟨proof⟩

lemma lookup-union3:
assumes k \notin \text{fmdom } xs
shows lookup (xs ++ f \ ys) k = lookup ys k
⟨proof⟩ including fset.lifting
⟨proof⟩

end

3 General purpose definitions and lemmas

theory JHelper imports
  Main
begin

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

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

lemma nth-in-set:
[\[ i < \text{length } xs ; xs ! i = x \]] \implies x \in \text{set } xs
⟨proof⟩

lemma disjI \[\text{intro}\]:
assumes \neg P \implies Q
shows P \lor Q
⟨proof⟩

lemma empty-eq-Plus-conv:
(\{\} = A \leftrightarrow B) = (A = \{\} \land B = \{\})
⟨proof⟩

3.1 Projection functions on triples

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

definition snd3 :: \'(a \times 'b \times 'c) \Rightarrow 'b
where snd3 \equiv \text{fst} \circ \text{snd}
definition thd3 :: 'a × 'b × 'c ⇒ 'c
where thd3 ≡ snd ○ snd

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

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

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

lemma tripleI:
fixes T U
assumes fst3 T = fst3 U
and snd3 T = snd3 U
and thd3 T = thd3 U
shows T = U
(proof)

end

4 Proof chains

theory Proofchain imports JHelper
begin

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

datatype ('a,'c) chain =
cNil 'a
| cCons 'a 'c ('a,'c) chain (\(\_{\_} \cdot \_ \cdot \_ \cdot [0,0,0] \ 60\))

For example, \(\{ a \} \cdot \text{proof} \cdot \{ \text{chain} \} \cdot \text{might} \cdot \{ \text{look} \} \cdot \text{like} \cdot \{ \text{this} \} \).

4.1 Projections

Project first assertion.

fun
\(\text{pre} :: ('a,'c) \text{ chain} ⇒ 'a\)
where
\[ \text{pre} \{ \emptyset \ P \} = P \]
\[ \text{pre} \{ \emptyset \ P \} \cdot \cdot \cdot = P \]

Project final assertion.

fun post :: (\texttt{'a,'c}) chain \Rightarrow \texttt{'a}
where
\[ \text{post} \{ \emptyset \ P \} = P \]
\[ \text{post} (\{ \emptyset \} \cdot \cdot \cdot) = \text{post} \Pi \]

Project list of commands.

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

4.2 Chain length

fun chainlen :: (\texttt{'a,'c}) chain \Rightarrow \texttt{nat}
where
\[ \text{chainlen} \{ \emptyset \} = 0 \]
\[ \text{chainlen} (\{ \emptyset \} \cdot \cdot \cdot) = 1 + (\text{chainlen} \Pi) \]

lemma len-comlist-chainlen:
\[ \text{length} (\text{comlist} \Pi) = \text{chainlen} \Pi \]
⟨proof⟩

4.3 Extracting triples from chains

\( \text{nthtriple} \Pi \ n \) extracts the \( n \)th triple of \( \Pi \), counting from 0. The function is well-defined when \( n < \text{chainlen} \Pi \).

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

The list of middle components of \( \Pi \)'s triples is the list of \( \Pi \)'s commands.

lemma snds-of-triples-form-comlist:
\[ \text{fixes} \ \Pi \ i \]
\[ \text{shows} \ i < \text{chainlen} \Pi \ \Rightarrow \ \text{snd}3 (\text{nthtriple} \Pi \ i) = (\text{comlist} \Pi)!i \]
⟨proof⟩

4.4 Evaluating a predicate on each triple of a chain

\( \text{chain-all} \ \varphi \) holds of \( \Pi \) iff \( \varphi \) holds for each of \( \Pi \)'s individual triples.
fun chain-all :: (('a × 'c × 'a) ⇒ bool) ⇒ ('a,'c) chain ⇒ bool

where
  chain-all ϕ (σ, x · Π) = (ϕ (σ, x, pre Π) ∧ chain-all ϕ Π)

lemma chain-all-mono [mono]:
  x ≤ y ⇒ chain-all x ≤ chain-all y
⟨proof⟩

lemma chain-all-nthtriple:
  (chain-all ϕ Π) = (∀ i < chainlen Π. ϕ (nthtriple Π i))
⟨proof⟩

4.5 A map function for proof chains

chainmap f g Π maps f over each of Π’s assertions, and g over each of Π’s commands.

fun chainmap :: ('a ⇒ 'b) ⇒ ('c ⇒ 'd) ⇒ ('a,'c) chain ⇒ ('b,'d) chain

where
  chainmap f g (σ, x · Π) = (σ, f x · chainmap f g Π

Mapping over a chain preserves its length.

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

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

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

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

4.6 Extending a chain on its right-hand side

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

where
  cSnoc (σ · y · τ) = (σ · y) · (σ · τ)
\[ cSnoc \{ \sigma \} \cdot x \cdot y \tau = \{ \sigma \} \cdot x \cdot (cSnoc y \tau) \]

**lemma** len-snoc:
- **fixes** \( y \) \( x \) \( P \)
- **shows** \( \text{chainlen} (cSnoc y x P) = 1 + (\text{chainlen} y \tau) \)

**lemma** pre-snoc:
- \( \text{pre} (cSnoc y x P) = \text{pre} y \tau \)

**lemma** post-snoc:
- \( \text{post} (cSnoc y x P) = P \)

**lemma** comlist-snoc:
- \( \text{comlist} (cSnoc y x p) = \text{comlist} y \tau @ [x] \)

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
  \( \text{Emp} :: \text{assertion} \)

axiomatization
  \( \text{Star} :: \text{assertion} \Rightarrow \text{assertion} \Rightarrow \text{assertion} \infixr \ast \ 55 \)

where
  \( \text{star-comm}: p \ast q = q \ast p \) and
\( \text{star-assoc: } (p \ast q) \ast r = p \ast (q \ast r) \) and
\( \text{star-emp: } p \ast \text{Emp} = p \) and
\( \text{emp-star: } \text{Emp} \ast p = p \)

lemma \( \text{star-rot: } q \ast p \ast r = p \ast q \ast r \)

\( \langle \text{proof} \rangle \)

axiomatization
\[ \text{Exists :: string } \Rightarrow \text{assertion } \Rightarrow \text{assertion} \]

Extracting the set of program variables mentioned in an assertion.

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

5.2 Commands

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

typedecl \text{command}

axiomatization
\[ \text{Choose :: command } \Rightarrow \text{command } \Rightarrow \text{command} \]

axiomatization
\[ \text{Loop :: command } \Rightarrow \text{command} \]

axiomatization
\[ \text{Skip :: command} \]

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

Extracting the set of program variables read by a command.

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

Extracting the set of program variables written by a command.
axiomatization

\[ \text{wr-com :: command} \Rightarrow \text{string set} \]

where

- \( \text{wr-com-choose: wr-com \ (\text{Choose } c_1 c_2) = wr-com \ c_1 \cup wr-com \ c_2} \)
- \( \text{and wr-com-loop: wr-com \ (\text{Loop } c) = wr-com \ c} \)
- \( \text{and wr-com-skip: wr-com \ (\text{Skip}) = \{\}} \)
- \( \text{and wr-com-seq: wr-com \ (c_1 ;; c_2) = wr-com \ c_1 \cup wr-com \ c_2} \)

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

\[ \text{prov-triple :: assertion} \times \text{command} \times \text{assertion} \Rightarrow \text{bool} \]

where

- \( \text{exists: prov-triple \ (p, c, q) \Rightarrow prov-triple \ (\exists x \ p, c, \exists x \ q)} \)
- \( \text{choose: \left[ \ \text{prov-triple \ (p, c_1, q); prov-triple \ (p, c_2, q)} \ \right]} \)
  \( \Rightarrow \text{prov-triple \ (p, \text{Choose } c_1 c_2, q)} \)
- \( \text{loop: prov-triple \ (p, c, p) \Rightarrow prov-triple \ (p, \text{Loop } c, p)} \)
- \( \text{frame: \left[ \ \text{prov-triple \ (p, c, q); wr-com(c) \cap rd-ass(r) = \{\}} \ \right]} \)
  \( \Rightarrow \text{prov-triple \ (p \ast r, c, q \ast r)} \)
- \( \text{skip: prov-triple \ (p, \text{Skip, p})} \)
- \( \text{seq: \left[ \ \text{prov-triple \ (p, c_1, q); prov-triple \ (q, c_2, r)} \ \right]} \)
  \( \Rightarrow \text{prov-triple \ (p, c_1 ;; c_2, r)} \)

Here are some derived proof rules, which are used in our ribbon-checking tool.

lemma choice-lemma:

\[ \text{assumes prov-triple \ (p_1, c_1, q_1) and prov-triple \ (p_2, c_2, q_2)} \]
\[ \text{and } p = p_1 \text{ and } p_1 = p_2 \text{ and } q = q_1 \text{ and } q_1 = q_2 \]
shows \( \text{prov-triple \ (p, \text{Choose } c_1 c_2, q)} \)
(\text{proof})

lemma loop-lemma:

\[ \text{assumes prov-triple \ (p_1, c, q_1) and } p = p_1 \text{ and } p_1 = q_1 \text{ and } q_1 = q \]
s shows \( \text{prov-triple \ (p, \text{Loop } c, q)} \)
(\text{proof})

lemma seq-lemma:

\[ \text{assumes prov-triple \ (p_1, c_1, q_1) and prov-triple \ (p_2, c_2, q_2)} \]
\[ \text{and } q_1 = p_2 \]
shows \( \text{prov-triple \ (p_1, c_1 ;; c_2, q_2)} \)
(\text{proof})

end
6 Ribbon proof interfaces

theory Ribbons-Interfaces imports
  Ribbons-Basic
  Proofchain
  HOL-Library.FSet
begin

Interfaces are the top and bottom boundaries through which diagrams can be connected into a surrounding context. For instance, when composing two diagrams vertically, the bottom interface of the upper diagram must match the top interface of the lower diagram.

We define a datatype of concrete interfaces. We then quotient by the associativity, commutativity and unity properties of our horizontal-composition operator.

6.1 Syntax of interfaces

datatype conc-interface =
  Ribbon-conc assertion
| HComp-int-conc conc-interface conc-interface (infix \otimes 50)
| Emp-int-conc (\varepsilon_c)
| Exists-int-conc string conc-interface

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

inductive equiv-int :: conc-interface \Rightarrow conc-interface \Rightarrow bool (infix \simeq 45)
where
  refl: P \simeq P
| sym: P \simeq Q \Rightarrow Q \simeq P
| trans: [P \simeq Q; Q \simeq R] \Rightarrow P \simeq R
| cong-hcomp1: P \simeq Q \Rightarrow P' \otimes_c P \simeq P' \otimes_c Q
| cong-hcomp2: P \simeq Q \Rightarrow P \otimes_c P' \simeq Q \otimes_c P'
| cong-exists: P \simeq Q \Rightarrow Exists-int-conc x P \simeq Exists-int-conc x Q
| hcomp-conc-assoc: P \otimes_c (Q \otimes_c R) \simeq (P \otimes_c Q) \otimes_c R
| hcomp-conc-comm: P \otimes_c Q \simeq Q \otimes_c P
| hcomp-conc-unit1: \varepsilon_c \otimes_c P \simeq P
| hcomp-conc-unit2: P \otimes_c \varepsilon_c \simeq P

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

quotient-type interface = conc-interface / equiv-int  
(proof)
lift-definition
Ribon :: assertion ⇒ interface
is Ribon-conc ⟨proof⟩

lift-definition
Emp-int :: interface (ε)
is εc ⟨proof⟩

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

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

lemma hcomp-comm:
(P ⊗ Q) = (Q ⊗ P)
⟨proof⟩

lemma hcomp- Assoc:
(P ⊗ (Q ⊗ R)) = ((P ⊗ Q) ⊗ R)
⟨proof⟩

lemma emp-hcomp:
ε ⊗ P = P
⟨proof⟩

lemma hcomp- emp:
P ⊗ ε = P
⟨proof⟩

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

6.2 An iterated horizontal-composition operator

definition iter-hcomp :: (′a fset) ⇒ (′a ⇒ interface) ⇒ interface
where
iter-hcomp X f ≡ ffold ((⊗) o f) ε X

syntax iter-hcomp-syntax ::
′a ⇒ (′a fset) ⇒ (′a ⇒ interface) ⇒ interface
((⊗) ·∈· ·) [0,0,10] 10
translations ⊗ x|∈|M. e == CONST iter-hcomp M (λx. e)
term $\bigotimes P \in \{Ps, f P\}$. $f P$ — this is eta-expanded, so prints in expanded form

term $\bigotimes P \in \{Ps, f\}$. $f$ — 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 $(\bigotimes v \in \text{vs. } \varphi \ v) = (\bigotimes v \in \text{vs. } \varphi' \ v)$
  ⟨proof⟩

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

lemma iter-hcomp-insert:
  assumes $v \not\in \text{vs}$
  shows $((\bigotimes x \in \text{finsert v ws, p x}) = (p v \otimes (\bigotimes x \in \text{ws, p x})$)
  ⟨proof⟩

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

6.3 Semantics of interfaces

The semantics of an interface is an assertion.

fun
conc-asn :: conc-interface ⇒ assertion
where
conc-asn (Ribbon-conc p) = p
| conc-asn (P ⊗ Q) = (conc-asn P) ⋆ (conc-asn Q)
| conc-asn (εc) = Emp
| conc-asn (Exists-int-conc x P) = Exists x (conc-asn P)

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

6.4 Program variables mentioned in an interface.

fun
\textbf{rd-conc-int} :: conc-interface ⇒ string set

\textbf{where}
\begin{align*}
\text{rd-conc-int} & (\text{Ribbon-conc } p) = \text{rd-ass } p \\
\text{rd-conc-int} & (P \otimes_c Q) = \text{rd-conc-int } P \cup \text{rd-conc-int } Q \\
\text{rd-conc-int} & (\epsilon_c) = \{} \\
\text{rd-conc-int} & (\exists x \text{-conc } P) = \text{rd-conc-int } P
\end{align*}

\textbf{lift-definition}
\textbf{rd-int} :: interface ⇒ string set
\textbf{is} \text{rd-conc-int}
\langle \text{proof} \rangle

The program variables read by an interface are the same as those read by its corresponding assertion.

\textbf{lemma} \text{rd-int-is-rd-ass:}
\text{rd-ass} (\text{asn } P) = \text{rd-int } P
\langle \text{proof} \rangle

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

\textbf{lemma} \text{seq-fold:}
\begin{align*}
\forall II. & \left[ \text{length } cs = \text{chainlen } II ; \ p1 = \text{asn } (\text{pre } II) ; \ p2 = \text{asn } (\text{post } II) ; \\
\forall i. \ i < \text{chainlen } II & \implies \text{prov-triple} \\
\text{asn} & (\text{fst3} (\text{nthtriple } II i)) , \ cs ! i , \ \text{asn} (\text{thd3} (\text{nthtriple } II i)) \right] \\
& \implies \text{prov-triple} (p1 , \ \text{foldr} (;) cs \text{ Skip} , p2)
\langle \text{proof} \rangle
\end{align*}
\textbf{end}

7 Syntax and proof rules for stratified diagrams

\textbf{theory} \text{Ribbons-Stratified} \textbf{imports}
\text{Ribbons-Interfaces}
\text{Proofchain}
\textbf{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

\textbf{datatype} \text{sdiagram} = SDiagram (\text{cell} \times \text{interface}) \text{ list}
\textbf{and} \text{cell} =
\begin{align*}
& \text{Filler interface} \\
& \text{Basic interface command interface} \\
& \text{Exists-sdia string sdiagram} \\
& \text{Choose-sdia interface sdiagram sdiagram interface} \\
& \text{Loop-sdia interface sdiagram interface}
\end{align*}
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 ϱ s) = foldr (;;) (map (com-cell ◦ fst) ϱ) 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 ϱ s) = (⋃ r ∈ set ϱ. wr-cell (fst r))
  wr-cell (Filler P) = {}
  wr-cell (Basic P c Q) = wr-com c
  wr-cell (Exists-sdia x D) = wr-sdia D
  wr-cell (Choose-sdia P D E Q) = wr-sdia D ∪ wr-sdia E
  wr-cell (Loop-sdia P D Q) = wr-sdia D

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

lemma wr-sdia-is-wr-com:
  fixes ϱ s :: row list
  and γ :: row
  shows (wr-sdia D = wr-com (com-sdia D))
  and (wr-cell γ = wr-com (com-cell γ))
  and (⋃ γ ∈ set ϱ. wr-cell (fst γ))
      = wr-com (foldr (;;) (map (λ(γ,F). com-cell γ) ϱ) Skip)
  and wr-cell (fst γ) = wr-com (com-cell (fst γ))
⟨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
\[ S\text{Basic}: \text{prov-triple} (\text{asn } P, c, \text{asn } Q) \implies \text{prov-cell} (\text{Basic } P c Q) P Q \]

\[ S\text{Exists}: \text{prov-sdia} D P Q \]
\[ \implies \text{prov-cell} (\text{Exists-sdia } x D) (\text{Exists-int } x P) (\text{Exists-int } x Q) \]

\[ S\text{Choice}: [ \text{prov-sdia} D P Q ; \text{prov-sdia} E P Q ] \]
\[ \implies \text{prov-cell} (\text{Choose-sdia } P D E Q) P Q \]

\[ S\text{Loop}: \text{prov-sdia} D P P \implies \text{prov-cell} (\text{Loop-sdia } D P P) P P \]

\[ S\text{Row}: [ \text{prov-cell } \gamma P Q ; \text{wr-cell } \gamma \cap \text{rd-int } F = \{ \} ] \]
\[ \implies \text{prov-sdia} (\text{SDiagram} (\text{comlist } \Pi)) (\text{pre } \Pi) (\text{post } \Pi) \]

7.3 Soundness

**lemma** soundness-strat-helper:
\[
(\text{prov-sdia } D P Q \implies \text{prov-triple} (\text{asn } P, \text{com-sdia } D, \text{asn } Q)) \land
(\text{prov-row } \varrho P Q \implies \text{prov-triple} (\text{asn } P, \text{com-cell} (\text{fst } \varrho), \text{asn } Q)) \land
(\text{prov-cell } \gamma P Q \implies \text{prov-triple} (\text{asn } P, \text{com-cell } \gamma, \text{asn } Q))
\]
\[
\langle \text{proof} \rangle
\]

**corollary** soundness-strat:

assumes \text{prov-sdia } D P Q

shows \text{prov-triple} (\text{asn } P, \text{com-sdia } D, \text{asn } Q)

\[
\langle \text{proof} \rangle
\]

end

8 Syntax and proof rules for graphical diagrams

theory Ribbons-Graphical imports
Ribbons-Interfaces begin

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

8.1 Syntax of graphical diagrams

Fix a type for node identifiers

typedecl node

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

datatype assertion-gadget =
Rib assertion
| Exists-dia string diagram
and command-gadget =
  Com command
\mid Choose-dia diagram diagram
\mid Loop-dia diagram

and diagram = Graph
  node fset
  node ⇒ assertion-gadget
  (node fset \times command-gadget \times node fset) list

type-synonym labelling = node ⇒ assertion-gadget

type-synonym edge = node fset \times command-gadget \times node fset

Projecting components from a graph

fun vertices :: diagram ⇒ node fset \( \cdot^* V \ [1000] 1000 \)
where (Graph V \& E) \( \cdot^* V = V \)

fun labelling :: diagram ⇒ labelling \( \cdot^* \Lambda \ [1000] 1000 \)
where (Graph V \& E) \( \cdot^* \Lambda = \Lambda \)

fun edges :: diagram ⇒ edge list \( \cdot^* E \ [1000] 1000 \)
where (Graph V \& E) \( \cdot^* E = E \)

8.2 Well formedness of graphical diagrams

definition acyclicity :: edge list ⇒ bool
where
  acyclicity E \equiv \text{acyclic} \left( \bigcup_{e \in \text{set } E} \text{fset} \ (\text{fst3 } e) \times \text{fset} \ (\text{thd3 } e) \right)

definition linearity :: edge list ⇒ bool
where
  linearity E \equiv
  \text{distinct } E \land \forall e \in \text{set } E. \forall f \in \text{set } E. e \neq f \rightarrow
  \text{fst3 } e \mid \text{fst3 } f = \{||\} \land
  \text{thd3 } e \mid \text{thd3 } f = \{||\}

lemma linearityD:
  assumes linearity E
  shows distinct E
  and \\ \forall e, f. \left[ e \in \text{set } E ; f \in \text{set } E ; e \neq f \right] \rightarrow
  \text{fst3 } e \mid \text{fst3 } f = \{||\} \land
  \text{thd3 } e \mid \text{thd3 } f = \{||\}
  \langle\text{proof}\rangle

lemma linearityD2:
  linearity E \implies \forall e, f. e \in \text{set } E \land f \in \text{set } E \land e \neq f \rightarrow
  \text{fst3 } e \mid \text{fst3 } f = \{||\} \land
  \text{thd3 } e \mid \text{thd3 } f = \{||\}
  \langle\text{proof}\rangle
inductive
\[\text{wf-ass} :: \text{assertion-gadget} \Rightarrow \text{bool} \quad \text{and} \]
\[\text{wf-com} :: \text{command-gadget} \Rightarrow \text{bool} \quad \text{and} \]
\[\text{wf-dia} :: \text{diagram} \Rightarrow \text{bool} \]
where
\[\text{wf-rib}: \text{wf-ass} (\text{Rib } p) \]
\[\text{wf-exists}: \text{wf-dia } G \Rightarrow \text{wf-ass } (\text{Exists-dia } x \ G) \]
\[\text{wf-com}: \text{wf-com } (\text{Com } c) \]
\[\text{wf-choice}: [ \text{wf-dia } G ; \text{wf-dia } H ] \Rightarrow \text{wf-com } (\text{Choose-dia } G \ H) \]
\[\text{wf-loop}: \text{wf-dia } G \Rightarrow \text{wf-com } (\text{Loop-dia } G) \]
\[\text{wf-dia}: [ \forall e \in \text{set } E. \text{wf-com } (\text{snd3 } e) ; \forall v \in \text{fset } V. \text{wf-ass } (\Lambda \ v) ; \]
\[\text{acyclicity } E ; \text{linearity } E ; \forall e \in \text{set } E. \text{fst3 } e \mid \cup \mid \text{thd3 } e \mid \subseteq \mid V \] \[\Rightarrow \text{wf-dia } (\text{Graph } V \ \Lambda \ E) \]

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

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

8.3 Initial and terminal nodes

definition \\
\text{initials} :: \text{diagram} \Rightarrow \text{node fset} \\
where \\
\text{initials } G = \text{ffilter } (\lambda v. (\forall e \in \text{set } G \cdot E. \ v \mid \notin \mid \text{thd3 } e)) \ G \cdot V 

definition \\
\text{terminals} :: \text{diagram} \Rightarrow \text{node fset} \\
where \\
\text{terminals } G = \text{ffilter } (\lambda v. (\forall e \in \text{set } G \cdot E. \ v \mid \notin \mid \text{fst3 } e)) \ G \cdot V 

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

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

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

8.4 Top and bottom interfaces

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

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

8.5 Proof rules for graphical diagrams

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

inductive-cases main-inv: prov-dia (Graph V Λ E) P Q
inductive-cases loop-inv: prov-com (Loop-dia G) P Q
inductive-cases choice-inv: prov-com (Choose-dia G H) P Q
inductive-cases basic-inv: prov-com (Com c) P Q
inductive-cases exists-inv: prov-ass (Exists-dia x G)
inductive-cases skip-inv: prov-ass (Rib p)
8.6 Extracting commands from diagrams

**Type-synonym** \( \text{lin} = (\text{node} + \text{edge}) \text{ list} \)

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

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

where

\[
\text{lins } G \equiv \{ \pi :: \text{lin}. \quad (\text{distinct } \pi) \\
\land (\text{set } \pi = (\text{fset } G \cdot V) <\to (\text{set } G \cdot E)) \\
\land (\forall i j v e. \ i < \text{length } \pi \land j < \text{length } \pi \land \pi!i = \text{Inl } v \land \pi!j = \text{Inr } e \\
\land v \in [\text{fst3 } e \to i < j) \\
\land (\forall j k w e. \ j < \text{length } \pi \land k < \text{length } \pi \land \pi!j = \text{Inr } e \land \pi!k = \text{Inl } w \land w \in [\text{thd3 } e \to j < k) \}
\]

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

assumes \( \pi \in \text{lins } G \)

shows \( (\text{distinct } \pi) \) and \( (\text{set } \pi = (\text{fset } G \cdot V) <\to (\text{set } G \cdot E)) \) and \( (\forall i j v e. \ i < \text{length } \pi \land j < \text{length } \pi \land \pi!i = \text{Inl } v \land \pi!j = \text{Inr } e \\
\land i < j) \) and \( (\forall j k w e. \ j < \text{length } \pi \land k < \text{length } \pi \land \pi!j = \text{Inr } e \land \pi!k = \text{Inl } w \land w < j < k) \)

**⟨Proof⟩**

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

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

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

**⟨Proof⟩**

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

**Inductive**

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

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

monos
coms-mono-helper

inductive-cases coms-skip-inv: coms-ass (Rib p) c
inductive-cases coms-exists-inv: coms-ass (Exists-dia x G) c
inductive-cases coms-basic-inv: coms-com (Com c′) c
inductive-cases coms-choice-inv: coms-com (Choose-dia G H) c
inductive-cases coms-loop-inv: coms-com (Loop-dia G) c
inductive-cases coms-main-inv: coms-dia G c

end

9 Soundness for graphical diagrams

theory Ribbons-Graphical-Soundness imports
  Ribbons-Graphical
  More-Finite-Map
begin

We prove that the proof rules for graphical ribbon proofs are sound with respect to the rules of separation logic.

We impose an additional assumption to achieve soundness: that the Frame rule has no side-condition. This assumption is reasonable because there are several separation logics that lack such a side-condition, such as “variables-as-resource”.

We first describe how to extract proofchains from a diagram. This process is similar to the process of extracting commands from a diagram, which was described in Ribbon-Proofs.Ribbons-Graphical. When we extract a proofchain, we don’t just include the commands, but the assertions in between them. Our main lemma for proving soundness says that each of these proofchains corresponds to a valid separation logic proof.
9.1 Proofstate chains

When extracting a proofchain from a diagram, we need to keep track of which nodes we have processed and which ones we haven’t. A proofstate, defined below, maps a node to “Top” if it hasn’t been processed and “Bot” if it has.

datatype topbot = Top | Bot

type-synonym proofstate = node \rightarrow 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) \rightarrow chain

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

fun next-ps :: proofstate ⇒ node + edge ⇒ proofstate
where
next-ps σ (Inl v) = σ \ominus \{\{v\}\} ++ f [\{v\} |⇒ Bot]
| next-ps σ (Inr e) = σ \ominus \text{fst3 } e ++ f [\text{thd3 } e |⇒ Top]

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

definition mk-ps-chain :: [ps-chain, (node + edge) list] ⇒ ps-chain
where
mk-ps-chain ≡ foldl (λΠ x. cSnoc Π x (next-ps (post Π) x)) π

lemma mk-ps-chain-preserves-length:
fixes Π π
shows chainlen (mk-ps-chain Π π) = chainlen Π + length π
⟨proof⟩

Distributing mk-ps-chain over (♯).

lemma mk-ps-chain-cons:
mk-ps-chain Π (x # π) = mk-ps-chain (cSnoc Π x (next-ps (post Π) x)) π
⟨proof⟩

Distributing mk-ps-chain over snoc.

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

Distributing \( mk-ps-chain \) over \( c\text{Cons} \).

**lemma** \( mk-ps-chain-ccons \):

\[
\text{fixes } \pi \Pi \\
\text{shows } mk-ps-chain (\qe \sigma \cdot x \cdot \Pi) \pi = \qe \sigma \cdot x \cdot mk-ps-chain \Pi \pi
\]

**lemma** \( pre-mk-ps-chain \):

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

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

**lemma** \( comlist-mk-ps-chain \):

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

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

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

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

**definition** \( \text{lins2} :: [\text{node fset, diagram}] \Rightarrow \text{lin set} \)

where

\[
\text{lins2} S G \equiv \{ \pi :: \text{lin} . \quad \text{(distinct } \pi) \} \\
\land (\text{set } \pi = (\text{fset } G^V - \text{fset } S) <\leftrightarrow \text{ set } G^E) \\
\land (\forall i j v e. i < \text{length } \pi \land j < \text{length } \pi \land \pi!i = \text{Inl } v \land \pi!j = \text{Inr } e \land v \in [\text{fst} e \rightarrow i < j) \\
\land (\forall j k w e. j < \text{length } \pi \land k < \text{length } \pi \land \pi!j = \text{Inr } e \land \pi!k = \text{Inl } w \land w \in [\text{thd} e \rightarrow j < k) \}
\]
lemma lins2D:
assumes π ∈ lins2 S G
shows distinct π
and set π = (fset G'V - fset S) <+> set G'E
and \( \forall i j v. (i < length π \land j < length π) \Rightarrow \pi ! i = Inl v \land \pi ! j = Inr e \land v \in \mathit{fst3} e \) \( \Rightarrow i < j \)
and \( \forall i k w. (j < length π \land k < length π) \Rightarrow \pi ! j = Inr e \land \pi ! k = Inl w \land w \in \mathit{thd3} e \) \( \Rightarrow j < k \)
⟨proof⟩

lemma lins2I:
assumes distinct π
and set π = (fset G'V - fset S) <+> set G'E
and \( \forall i j v. (i < length π \land j < length π) \Rightarrow \pi ! i = Inl v \land \pi ! j = Inr e \land v \in \mathit{fst3} e \) \( \Rightarrow i < j \)
and \( \forall j k w. (j < length π \land k < length π) \Rightarrow \pi ! j = Inr e \land \pi ! k = Inl w \land w \in \mathit{thd3} e \) \( \Rightarrow j < k \)
shows π ∈ lins2 S G
⟨proof⟩

When S is empty, the two definitions coincide.

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

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

definition initial-ps :: diagram ⇒ proofstate
where
\( \text{initial-ps } G \equiv \left[ \text{initials } G \Rightarrow \text{Top} \right] \)

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

definition initial-ps2 :: [node fset, diagram] ⇒ proofstate
where
\( \text{initial-ps2 } S G \equiv \left[ \text{initials } G - S \Rightarrow \text{Top} \right] + + f \left[ S \Rightarrow \text{Bot} \right] \)

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 ps-chains :: diagram ⇒ ps-chain set
where
\[ ps-chains G \equiv mk-ps-chain (cNil (initial-ps G)) \cdot lins G \]

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

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

**where**
\[ ps-chains2 S G \equiv mk-ps-chain (cNil (initial-ps2 S G)) \cdot lins2 S G \]

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

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

We now wish to describe proofstates chain that are well-formed. First, let us say that \( f \uplus f \) disjoint \( g \) is defined, when \( f \) and \( g \) have disjoint domains, as \( f \uplus f \) \( g \).

Then, a well-formed proofstate chain consists of triples of the form \((\sigma \uplus f \text{ disjoint } [\{v\}] \Rightarrow \text{Top }), \text{Inl } v, \sigma \uplus f \text{ disjoint } [\{v\}] \Rightarrow \text{Bot }\), where \( v \) is a node, or of the form \((\sigma \uplus f \text{ disjoint } [\{ws\}] \Rightarrow \text{Bot }), \text{Inr } e, \sigma \uplus f \text{ disjoint } [\{ws\}] \Rightarrow \text{Top }\), where \( e \) is an edge with source and target nodes \( vs \) and \( ws \) respectively.

The definition below describes a well-formed triple; we then lift this to complete chains shortly.

**definition**
\[ wf-ps-triple :: \text{proofstate} \times (\text{node + edge}) \times \text{proofstate} \Rightarrow \text{bool} \]

**where**
\[ wf-ps-triple T = (\text{case snd3 } T \text{ of} \]
\[ \quad \text{Inl } v \Rightarrow (\exists \sigma. v \notin fmdom \sigma \land \sigma_1 = [\{v\}] \Rightarrow \text{Top } \uplus f \sigma \land \sigma_2 = [\{v\}] \Rightarrow \text{Bot } \uplus f \sigma) \]
\[ \quad \text{Inr } e \Rightarrow (\exists \sigma. (\text{fst3 } e \cup \text{thd3 } e) \cap fmdom \sigma = \{||\} \land \sigma_1 = [\text{fst3 } e \Rightarrow \text{Bot } \uplus f \sigma \land \sigma_2 = [\text{thd3 } e \Rightarrow \text{Top } \uplus f \sigma) \]) \]

**lemma** \( wf-ps-triple-nodeI \):
\[ \text{assumes } (\exists \sigma. v \notin fmdom \sigma \land \sigma_1 = [\{v\}] \Rightarrow \text{Top } \uplus f \sigma \land \sigma_2 = [\{v\}] \Rightarrow \text{Bot } \uplus f \sigma \]
\[ \text{shows } wf-ps-triple (\sigma_1, \text{Inl } v, \sigma_2) \]

**proof**

**lemma** \( wf-ps-triple-edgeI \):
\[ \text{assumes } (\exists \sigma. (\text{fst3 } e \cup \text{thd3 } e) \cap fmdom \sigma = \{||\}) \land \sigma_1 = [\text{fst3 } e \Rightarrow \text{Bot } \uplus f \sigma \land \sigma_2 = [\text{thd3 } e \Rightarrow \text{Top } \uplus f \sigma \]
\[ \text{shows } wf-ps-triple (\sigma_1, \text{Inr } e, \sigma_2) \]

26
\textbf{proof}

\textbf{definition}
\[\text{wf-ps-chain :: ps-chain } \Rightarrow \text{ bool}\]
\textbf{where}
\[\text{wf-ps-chain } \equiv \text{ chain-all wf-ps-triple}\]

\textbf{lemma} \textit{next-initial-ps2-vertex}:
\[
\text{initial-ps2 } (\{\{v\}\} \cup S) \ G = \text{initial-ps2 } S \ G \ominus \{\{v\}\} \ ++_f [\{\{v\}\}] \Rightarrow \text{Bot}\]
\textit{proof}

\textbf{lemma} \textit{next-initial-ps2-edge}:
\textbf{assumes} \[G = \text{Graph } V \Lambda E \text{ and } G' = \text{Graph } V' \Lambda E' \text{ and} \]
\textbf{assumes} \[V' = V - \text{fst3 } e \text{ and } E' = \text{removeAll } e \ E \text{ and } e \in \text{set } E \text{ and} \]
\textbf{assumes} \[\text{fst3 } e \subseteq S \text{ and } S \subseteq \text{initials } G \text{ and } \text{wf-dia } G\]
\textbf{shows} \[\text{initial-ps2 } (S - \text{fst3 } e) \ G' = \text{initial-ps2 } S \ G \ominus \text{fst3 } e \ ++_f [\text{thd3 } e] \Rightarrow \text{Top}\]
\textit{proof}

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

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

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 \(S\).

\textbf{lemmas} \[\text{[simp] = fmember.rep-eq}\]

\textbf{lemma} \textit{wf-chains2}:
\textbf{fixes} \(k\)
\textbf{assumes} \[S \subseteq \text{initials } G\]
\textbf{assumes} \[\text{wf-dia } G\]
\textbf{assumes} \[\Pi \in \text{ps-chains2 } S \ G\]

\textit{proof}
and \( \text{fcard } G \cdot V + \text{length } G \cdot E = k + \text{fcard } S \)

shows \( \text{wf-ps-chain } \Pi \land (\text{post } \Pi = [\text{terminals } G \Rightarrow \text{Bot }]) \)

\(\langle \text{proof} \rangle\)

corollary \( \text{wf-chains}: \)
- assumes \( \text{wf-dia } G \)
- assumes \( \Pi \in \text{ps-chains } G \)
- shows \( \text{wf-ps-chain } \Pi \land \text{post } \Pi = [\text{terminals } G \Rightarrow \text{Bot } ] \)

\(\langle \text{proof} \rangle\)

9.2 Interface chains

type-synonym \( \text{int-chain} = (\text{interface}, \text{assertion-gadget} + \text{command-gadget}) \cdot \text{chain} \)

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.

\(\text{definition ps-to-int } :: [\text{diagram}, \text{proofstate}] \Rightarrow \text{interface} \)

where

\[ \text{ps-to-int } G \sigma \equiv \bigotimes_v [\text{fdom } \sigma. \text{case-topbot top-ass bot-ass (lookup } \sigma v) (G \cdot \Lambda v) \] 

\(\text{definition ps-chain-to-int-chain } :: [\text{diagram}, \text{ps-chain}] \Rightarrow \text{int-chain} \)

where

\[ \text{ps-chain-to-int-chain } G \Pi \equiv \text{chainmap (ps-to-int } G) ((\text{case-sum (Inl } \circ G \cdot \Lambda) (\text{Inr } \circ \text{snd3}))) \Pi \]

\(\text{lemma ps-chain-to-int-chain-simp}: \)

\[ \text{ps-chain-to-int-chain } (\text{Graph } V \cdot \Lambda E) \Pi = \text{chainmap (ps-to-int } (\text{Graph } V \cdot \Lambda E)) ((\text{case-sum (Inl } \circ \Lambda) (\text{Inr } \circ \text{snd3}))) \Pi \]

\(\langle \text{proof} \rangle\)

9.3 Soundness proof

We assume that \( \text{wr-com} \) always returns \( \{\} \). This is equivalent to changing our axiomatization of separation logic such that the frame rule has no side-condition. One way to obtain a separation logic lacking a side-condition on its frame rule is to use variables-as-resource.

We proceed by induction on the proof rules for graphical diagrams. We show that: (1) if a diagram \( G \) is provable w.r.t. interfaces \( P \) and \( Q \), then \( P \) and \( Q \) are the top and bottom interfaces of \( G \), and that the Hoare triple \((\text{asn } P, c, \text{asn } Q)\) is provable for each command \( c \) that can be extracted
from $G$; (2) if a command-gadget $C$ is provable w.r.t. interfaces $P$ and $Q$, then the Hoare triple $(asn P, c, asn Q)$ 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: $\bigwedge c. \text{wr-com } c = {}$

**Shows**

$(\text{prov-dia } G \ P \ Q \rightarrow$

$(P = \text{top-dia } G \land Q = \text{bot-dia } G \land$

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

$\land (\text{prov-com } C \ P \ Q \rightarrow$

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

$\land (\text{prov-ass } A \rightarrow$

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

**Proof**

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

**Corollary soundness-graphical:**

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

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

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

**Proof**

**References**


