Ribbon Proofs for Separation Logic  
(Isabelle Formalisation)  

John Wickerson  
May 28, 2015  

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 Union 3  
  2.2 Difference 4  
  2.3 Comprehension 4  
  2.4 The empty finite partial function 4  
  2.5 Domain 4  
  2.6 Lookup 5  
3 General purpose definitions and lemmas 6  
  3.1 Projection functions on triples 7  
4 Proof chains 7  
  4.1 Projections 8  
  4.2 Chain length 8  
  4.3 Extracting triples from chains 8  
  4.4 Evaluating a predicate on each triple of a chain 9
1 Introduction

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

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

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

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

2 Finite partial functions

theory Finite-Map imports Main
~~~/src/HOL/Library/FSet
begin

The type of finite partial functions is obtained by restricting the type of partial functions to those with a finite domain. We use the lifting package to transfer several theories from the Map library to our finite setting.

typedef ('k, 'v) fmap (infix ->f) = {f :: 'k -> 'v. finite (dom f)}
⟨proof⟩

setup-lifting type-definition-fmap

2.1 Union

lift-definition fmap-add :: ('k ->f 'v) ⇒ ('k ->f 'v) ⇒ ('k ->f 'v)
is map-add
⟨proof⟩

abbreviation FMAP-ADD :: ('k ->f 'v) ⇒ ('k ->f 'v) ⇒ ('k ->f 'v) (infixl ⊕ 100)
where
xs ⊕ ys ≡ fmap-add xs ys

lemma fmap-add-assoc:
A ⊕ (B ⊕ C) = (A ⊕ B) ⊕ C
⟨proof⟩
2.2 Difference

**definition**
\[ \text{map-diff} :: (k 
\rightarrow \text{v}) \Rightarrow k \text{ fset} \Rightarrow (k \rightarrow \text{v}) \]

**where**
\[ \text{map-diff} k \text{ s} = \text{restrict-map} f (\neg \text{fset} \text{ s}) \]

**lift-definition**
\[ \text{fmap-diff} :: (k \rightarrow f \text{ v}) \Rightarrow k \text{ fset} \Rightarrow (k \rightarrow f \text{ v}) \]

**is** map-diff

**abbreviation**
\[ \text{FMAP-DIFF} :: (k \rightarrow f \text{ v}) \Rightarrow k \text{ fset} \Rightarrow (k \rightarrow f \text{ v}) \]

**where**
\[ xs \ominus ys \equiv \text{fmap-diff} xs ys \]

2.3 Comprehension

**definition**
\[ \text{make-map} :: k \text{ fset} \Rightarrow v \Rightarrow (k \rightarrow v) \]

**where**
\[ \text{make-map} k \text{ s} x = \lambda k. \text{if} \ k \in \text{fset} \text{ s} \text{ then} x \text{ else} \text{None} \]

**lemma** dom-make-map:
\[ \text{dom} (\text{make-map} k \text{ s} v) = \text{fset} \text{ s} \]

**lift-definition**
\[ \text{make-fmap} :: k \text{ fset} \Rightarrow v \Rightarrow (k \rightarrow f v) \]

**is** make-map

**abbreviation**
\[ \text{MAKE-FMAP} :: k \text{ fset} \Rightarrow v \Rightarrow (k \rightarrow f v) \]

**where**
\[ [ k \rightarrow v ] \equiv \text{make-fmap} k \text{ s} v \]

2.4 The empty finite partial function

**lift-definition**
\[ \text{fmap-empty} :: (k \rightarrow f \text{ v}) \]

**is** empty

**2.5 Domain**

**definition**
\[ \text{dom-fset} :: (k \rightarrow v) \Rightarrow k \text{ fset} \]

**where** dom-fset f = THE x. fset x = dom f
2.6 Lookup

lift-definition
   lookup :: ('k → f 'v) ⇒ 'k fset
is op ◦ the ⟨proof⟩

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

lemma lookup-make-fmap1:
lookup [ \{k\} |\Rightarrow v \} k = v
(proof)

lemma lookup-union1:
assumes k |\in| fdom ys
shows lookup (xs \oplus ys) k = lookup ys k
(proof)

lemma lookup-union2:
assumes k |\notin| fdom ys
shows lookup (xs \oplus ys) k = lookup xs k
(proof)

lemma lookup-union3:
assumes k |\notin| fdom xs
shows lookup (xs \oplus ys) k = lookup ys k
(proof)

end

3 General purpose definitions and lemmas

theory JHelper imports
Main
begin

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

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

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

lemma disjI [intro]:
assumes \neg P \implies Q
shows P \lor Q
(proof)

lemma empty-eq-Plus-conv:
(\{\} = A \leftrightarrow B) \equiv (A = \{\} \land B = \{\})
(proof)
3.1 Projection functions on triples

definition fst3 :: 'a × 'b × 'c ⇒ 'a
where fst3 ≡ fst

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

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

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

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

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

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

end

4 Proof chains

theory Proofchain imports JHelper begin

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

datatype ('a,'c) chain =
cNil 'a
| cCons 'a 'c ('a,'c) chain
⟨proof⟩·...· [0,0,0] 60

For example, {a} · proof · {chain} · might · {look} · like · {this}.

4.1 Projections

Project first assertion.

fun
pre :: ('a,'c) chain ⇒ 'a
where
pre [P] = P
| pre ([P] · · ·) = P

Project final assertion.

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

Project list of commands.

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

4.2 Chain length

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

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

4.3 Extracting triples from chains

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

fun
nthtriple :: ('a,'c) chain ⇒ nat ⇒ ('a * 'c * 'a)
where
nthtriple ([P] x · II) 0 = (P, x, pre II)
| nthtriple ([P] x · II) (Suc n) = nthtriple Π n

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

lemma snds-of-triples-form-comlist:
fixes Π i
shows \( i < \text{chainlen } \Pi \implies \text{snd3} \left( \text{nthtriple } \Pi \ i \right) = \left( \text{comlist } \Pi \right)!i \)  

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

### 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 \( :: \left( (\ 'a \times \ 'c \times \ 'a) \implies \text{bool} \right) \implies (\ 'a,\ 'c) \text{ chain} \implies \text{bool} \)

**where**

- chain-all \( \varphi \{ | \sigma \} = \text{True} \)
- chain-all \( \varphi \{ | \sigma \} \cdot x \cdot \Pi \) = (\( \varphi \ (\sigma,\text{pre } \Pi) \land \text{chain-all } \varphi \Pi \))

**lemma** chain-all-mono [mono]:

\( x \leq y \implies \text{chain-all } x \leq \text{chain-all } y \)

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

**lemma** chain-all-nthtriple:

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

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

### 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 \( :: (\ 'a \Rightarrow \ 'b) \Rightarrow (\ 'c \Rightarrow \ 'd) \Rightarrow (\ 'a,\ 'c) \text{ chain} \Rightarrow (\ 'b,\ 'd) \text{ chain} \)

**where**

- chainmap \( f \ g \{ | \ P \} = \{ | f \ P \} \)
- chainmap \( f \ g \ (\{ | \ P \} \cdot x \cdot \Pi) = \{ | f \ P \} \cdot g \cdot x \cdot \text{chainmap } f \ g \Pi \)

Mapping over a chain preserves its length.

**lemma** chainmap-preserves-length:

\( \text{chainlen } (\text{chainmap } f \ g \Pi) = \text{chainlen } \Pi \)

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

**lemma** pre-chainmap:

\( \text{pre } (\text{chainmap } f \ g \Pi) = f \ (\text{pre } \Pi) \)

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

**lemma** post-chainmap:

\( \text{post } (\text{chainmap } f \ g \Pi) = f \ (\text{post } \Pi) \)

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

**lemma** nthtriple-chainmap:

- assumes \( i < \text{chainlen } \Pi \)
- shows \( \text{nthtriple } (\text{chainmap } f \ g \Pi) \ i \)
  \( = (\lambda \ t. \ f \ (\text{fst3 } t), \ g \ (\text{snd3 } t), \ f \ (\text{thd3 } t)) \) (\( \text{nthtriple } \Pi \ i \))

\( \langle \text{proof} \rangle \)
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 · τ
| cSnoc (σ | x · ε) y τ = σ | x · (cSnoc y τ)

lemma \textit{len-snoc}:
  \textbf{fixes} \; ε \; x \; P
  \textbf{shows} \; \text{chainlen} (cSnoc ε x P) = 1 + (\text{chainlen} \; ε)
(\textbf{proof})

lemma \textit{pre-snoc}:
  \textbf{pre} (cSnoc ε x P) = \text{pre} \; ε
(\textbf{proof})

lemma \textit{post-snoc}:
  \textbf{post} (cSnoc ε x P) = P
(\textbf{proof})

lemma \textit{comlist-snoc}:
  \text{comlist} (cSnoc ε x p) = \text{comlist} \; ε @ [x]
(\textbf{proof})

end

5 Assertions, commands, and separation logic proof rules

\textbf{theory} \; \textit{Ribbons-Basic} \; \textbf{imports}
\textit{Main}
\begin{proof}

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 \texttt{emp} constant, a star-operator, and existentially-quantified logical variables.

typedecl \texttt{assertion}

axiomatization
\textit{Emp} :: assertion

\textbf{axiomatization}

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

\textbf{where}

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

\textbf{lemma} \textit{star-rot}:

q \ast p \ast r = p \ast q \ast r

\langle \text{proof} \rangle

\textbf{axiomatization}

\textit{Exists} :: string \Rightarrow assertion \Rightarrow assertion

Extracting the set of program variables mentioned in an assertion.

\textbf{axiomatization}

\textit{rd-ass} :: assertion \Rightarrow string set

\textbf{where}

\textit{rd-emp}: rd-ass \textit{Emp} = \{\}

\textbf{and} \textit{rd-star}: rd-ass (p \ast q) = rd-ass p \cup rd-ass q

\textbf{and} \textit{rd-exists}: rd-ass (\textit{Exists} x p) = rd-ass p

\textbf{5.2 Commands}

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

\textbf{typedec} \textit{command}

\textbf{axiomatization}

\textit{Choose} :: command \Rightarrow command \Rightarrow command

\textbf{axiomatization}

\textit{Loop} :: command \Rightarrow command

\textbf{axiomatization}

\textit{Skip} :: command

\textbf{axiomatization}

\textit{Seq} :: command \Rightarrow command \Rightarrow command (\textit{infixr} ;; 55)

\textbf{where}

\textit{seq-assoc}: c1 ;; (c2 ;; c3) = (c1 ;; c2) ;; c3

\textbf{and} \textit{seq-skip}: c ;; \textit{Skip} = c

\textbf{and} \textit{skip-seq}: \textit{Skip} ;; c = c

Extracting the set of program variables read by a command.

\textbf{axiomatization}

\textit{rd-com} :: command \Rightarrow 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 \( \text{rd-com-skip} : \text{rd-com} (\text{Skip}) = \emptyset \)
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**

\( \text{wr-com} :: \text{command} \Rightarrow \text{string set} \)

where \( \text{wr-com-choose} : \text{wr-com} (\text{Choose } c1 \ c2) = \text{wr-com } c1 \cup \text{wr-com } c2 \)
and \( \text{wr-com-loop} : \text{wr-com} (\text{Loop } c) = \text{wr-com } c \)
and \( \text{wr-com-skip} : \text{wr-com} (\text{Skip}) = \emptyset \)
and \( \text{wr-com-seq} : \text{wr-com} (c1 \ ;\ ; c2) = \text{wr-com } c1 \cup \text{wr-com } c2 \)

### 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**

\( \text{prov-triple} :: \text{assertion} \times \text{command} \times \text{assertion} \Rightarrow \text{bool} \)

where

- \( \text{exists} : \text{prov-triple} (p, c, q) \Rightarrow \text{prov-triple} (\text{Exists } x \ p, c, \text{Exists } x \ q) \)
- \( \text{choose} : [ \text{prov-triple} (p, c1, q); \text{prov-triple} (p, c2, q) ] \Rightarrow \text{prov-triple} (p, \text{Choose } c1 \ c2, q) \)
- \( \text{loop} : \text{prov-triple} (p, c, p) \Rightarrow \text{prov-triple} (p, \text{Loop } c, p) \)
- \( \text{frame} : [ \text{prov-triple} (p, c, q); \text{wr-com} (c) \cap \text{rd-ass}(r) = \emptyset ] \Rightarrow \text{prov-triple} (p \star r, c, q \star r) \)
- \( \text{skip} : \text{prov-triple} (p, \text{Skip}, p) \Rightarrow \text{prov-triple} (p, c1 \ ;\ ; c2, r) \)

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

**Lemma choice-lemma:**

- **Assumes** \( \text{prov-triple} (p1, c1, q1) \text{ and } \text{prov-triple} (p2, c2, q2) \)
- **And** \( p = p1 \text{ and } p1 = p2 \text{ and } q = q1 \text{ and } q1 = q2 \)
- **Shows** \( \text{prov-triple} (p, \text{Choose } c1 \ c2, q) \)

(\text{proof})

**Lemma loop-lemma:**

- **Assumes** \( \text{prov-triple} (p1, c, q1) \text{ and } p = p1 \text{ and } p1 = q1 \text{ and } q1 = q \)
- **Shows** \( \text{prov-triple} (p, \text{Loop } c, q) \)

(\text{proof})

**Lemma seq-lemma:**

- **Assumes** \( \text{prov-triple} (p1, c1, q1) \text{ and } \text{prov-triple} (p2, c2, q2) \)
- **And** \( q1 = p2 \)
- **Shows** \( \text{prov-triple} (p1, c1 \ ;\ ; c2, q2) \)
\langle proof \rangle

end

6 Ribbon proof interfaces

theory Ribbons-Interfaces imports
Ribbons-Basic
Proofchain
~/src/HOL/Library/FSet
begin

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

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

6.1 Syntax of interfaces
datatype conc-interface =
Ribbon-conc assertion
| HComp-int-conc conc-interface conc-interface (infix \( \otimes \))
| Emp-int-conc \( \varepsilon \)
| 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 \( \otimes \). The final
two make \( \varepsilon \) the left and right unit of \( \otimes \).

inductive equiv-int :: conc-interface \Rightarrow conc-interface \Rightarrow bool (infix \( \simeq \))
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: \( P \simeq Q \Rightarrow P' \otimes \varepsilon \simeq P' \otimes Q \)
cong-hcomp2: \( P \simeq Q \Rightarrow P \otimes \varepsilon \simeq P \otimes Q \)
cong-exists: \( P \simeq Q \Rightarrow \text{Exists-int-conc x P} \simeq \text{Exists-int-conc x Q} \)
hcomp-conc-assoc: \( P \otimes (Q \otimes \varepsilon R) \simeq (P \otimes Q) \otimes \varepsilon R \)
hcomp-conc-comm: \( P \otimes \varepsilon Q \simeq P \otimes Q \)
hcomp-conc-unit1: \( \varepsilon \otimes \varepsilon P \simeq P \)
hcomp-conc-unit2: \( P \otimes \varepsilon \varepsilon \simeq P \)

lemma equiv-int-cong-hcomp:
\[ P \simeq Q ; P' \simeq Q' \implies P \otimes_c P' \simeq Q \otimes_c Q' \]

\textbf{proof}

\textbf{quotient-type} interface = conc-interface / equiv-int

\textbf{lift-definition}

\textit{Ribbon} :: \textbf{assertion} \Rightarrow \textbf{interface}
\textit{is} Ribbon-conc \textbf{proof}

\textbf{lift-definition}

\textit{Emp-int} :: \textbf{interface} (\varepsilon)
\textit{is} \varepsilon_c \textbf{proof}

\textbf{lift-definition}

\textit{Exists-int} :: \textbf{string} \Rightarrow \textbf{interface} \Rightarrow \textbf{interface}
\textit{is} Exists-int-conc \textbf{proof}

\textbf{lift-definition}

\textit{HComp-int} :: \textbf{interface} \Rightarrow \textbf{interface} \Rightarrow \textbf{interface} (\textbf{infix} \otimes 50)
\textit{is} HComp-int-conc \textbf{proof}

\textbf{lemma} hcomp-comm:
\( (P \otimes Q) = (Q \otimes P) \)
\textbf{proof}

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

\textbf{lemma} emp-hcomp:
\( \varepsilon \otimes P = P \)
\textbf{proof}

\textbf{lemma} hcomp-emp:
\( P \otimes \varepsilon = P \)
\textbf{proof}

\textbf{lemma} comp-fun-commute-hcomp:
\textit{comp-fun-commute} \textbf{op} \textbf{proof}

6.2 An iterated horizontal-composition operator

\textbf{definition} iter-hcomp :: \textbf{('a fset)} \Rightarrow \textbf{('a \Rightarrow \textbf{interface})} \Rightarrow \textbf{interface}

\textbf{where}

\textit{iter-hcomp} \textbf{X} \textit{f} \equiv \textbf{ffold} \textbf{op} \circ \textbf{f} \varepsilon \textbf{X}
syntax iter-hcomp-syntax ::
'a ⇒ ('a fset) ⇒ ('a ⇒ interface) ⇒ interface
((⨂ x |∈| fset. -) [0,0,10] 10)
translations ⨂ x |∈| M. e ↔ CONST iter-hcomp M (λx. e)

term ⨂ P |∈| Ps. f P — this is eta-expanded, so prints in expanded form
term ⨂ P |∈| Ps. f — this isn’t eta-expanded, so prints as written

lemma iter-hcomp-cong:
  assumes ∀ v ∈ fset vs. ϕ v = ϕ' v
  shows (⨂ v |∈| vs. ϕ v) = (⨂ v |∈| vs. ϕ' v)
⟨proof⟩

lemma iter-hcomp-empty:
  shows (⨂ x |∈| {||}. p x) = ε
⟨proof⟩

lemma iter-hcomp-insert:
  assumes v |∉| us
  shows (⨂ x |∈| finsert v ws. p x) = (p v ⊗ (⨂ x |∈| ws. p x))
⟨proof⟩

lemma iter-hcomp-union:
  assumes vs |∩| us = {||}
  shows (⨂ x |∈| vs |∪| us. p x) = ((⨂ x |∈| vs. p x) ⊗ (⨂ x |∈| us. 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 (ε_c) = Emp
conc-asn (Exists-int-conc x P) = Exists x (conc-asn P)

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

lemma asn-simps [simp]:
asn (Ribbon p) = p
asn (P ⊗ Q) = (asn P) ∗ (asn Q)
asn ε = Emp
asn (Exists-int x P) = Exists x (asn P)
6.4 Program variables mentioned in an interface.

fun rd-conc-int :: conc-interface ⇒ string set
where rd-conc-int (Ribbon-conc p) = rd-ass p
| rd-conc-int (P ⊗ c Q) = rd-conc-int P ∪ rd-conc-int Q
| rd-conc-int (ε c) = {}
| rd-conc-int (Exists-int-conc x P) = rd-conc-int P

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

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

lemma rd-int-is-rd-ass:
rd-ass (asn P) = rd-int P
⟨proof⟩

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

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

7 Syntax and proof rules for stratified diagrams

theory Ribbons-Stratified imports
  Ribbons-Interfaces
  Proofchain
begin

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

7.1 Syntax of stratified diagrams

datatype sdiagram = SDiagram (cell × interface) list
and cell =
Filler interface | Basic interface command interface
| Exists-sdia string sdiagram | Choose-sdia interface sdiagram sdiagram interface | Loop-sdia interface sdiagram sdiagram interface

**datatype-compat** sdiagram cell

**type-synonym** row = cell × interface

Extracting the command from a stratified diagram.

**fun**
\[ \text{com-sdia} :: sdiagram ⇒ command \text{ and } \text{com-cell} :: cell ⇒ command \]

**where**
\[ \text{com-sdia} (SDiagram } ϱ \text{s} ) = \text{foldr} (\text{op };;) (\text{map} (\text{com-cell} ∘ \text{fst}) \text{gs}) \text{Skip} \]
\[ \text{com-cell} (\text{Filler } P) = \text{Skip} \]
\[ \text{com-cell} (\text{Basic } P \text{ c } Q) = c \]
\[ \text{com-cell} (\text{Exists-sdia } x \text{ D}) = \text{com-sdia} \text{D} \]
\[ \text{com-cell} (\text{Choose-sdia } P \text{ D } E \text{ Q}) = \text{Choose} (\text{com-sdia} \text{D}) (\text{com-sdia} \text{E}) \]
\[ \text{com-cell} (\text{Loop-sdia } P \text{ D } Q) = \text{Loop} (\text{com-sdia} \text{D}) \]

Extracting the program variables written by a stratified diagram.

**fun**
\[ \text{wr-sdia} :: sdiagram ⇒ string set \text{ and } \text{wr-cell} :: cell ⇒ string set \]

**where**
\[ \text{wr-sdia} (SDiagram } ϱ \text{s} ) = (\bigcup \text{r} ∈ \text{set} \text{gs}. \text{wr-cell} (\text{fst} \text{r})) \]
\[ \text{wr-cell} (\text{Filler } P) = \{\} \]
\[ \text{wr-cell} (\text{Basic } P \text{ c } Q) = \text{wr-com} \text{c} \]
\[ \text{wr-cell} (\text{Exists-sdia } x \text{ D}) = \text{wr-sdia} \text{D} \]
\[ \text{wr-cell} (\text{Choose-sdia } P \text{ D } E \text{ Q}) = \text{wr-sdia} \text{D} \cup \text{wr-sdia} \text{E} \]
\[ \text{wr-cell} (\text{Loop-sdia } P \text{ D } Q) = \text{wr-sdia} \text{D} \]

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

**lemma** \( \text{wr-sdia-is-wr-com} : \)
\[ \text{fixes } \text{gs :: row list and } \text{g :: row} \]
\[ \text{shows} (\text{wr-sdia } \text{D} = \text{wr-com} (\text{com-sdia} \text{D})) \]
\[ (\text{wr-cell } γ = \text{wr-com} (\text{com-cell } γ)) \]
\[ (\bigcup \text{g} ∈ \text{set} \text{gs}. \text{wr-cell} (\text{fst} \text{g})) \]
\[ = \text{wr-com} (\text{foldr} (\text{op };;) (\text{map} (λ(γ,F). \text{com-cell } γ) \text{gs}) \text{Skip}) \]
\[ \text{and } \text{wr-cell} (\text{fst} \text{g}) = \text{wr-com} (\text{com-cell} (\text{fst} \text{g})) \]
prov-sdia :: [sdiagram, interface, interface] ⇒ bool and
prov-row :: [row, interface, interface] ⇒ bool and
prov-cell :: [cell, interface, interface] ⇒ bool

where
SRibbon: prov-cell (Filler P) P P
| SBasic: prov-triple (asn P, c, asn Q) ⇒ prov-cell (Basic P c Q) P Q
| SExists: prov-sdia D P Q
⇒ prov-cell (Exists-sdia x D) (Exists-int x P) (Exists-int x Q)
| SChoice: [ prov-sdia D P Q ; prov-sdia E P Q ]
⇒ prov-cell (Choose-sdia P D E Q) P Q
| SRow: [ prov-cell γ P Q ; writ-cell γ ∩ rd-int F = {} ]
⇒ prov-row (γ, F) (P ⊗ F) (Q ⊗ F)
| SMain: [ chain-all (λ(P,q,Q). prov-row q P Q) Π ; 0 < chainlen Π ]
⇒ prov-sdia (SDiagram (comlist Π)) (pre Π) (post Π)

7.3 Soundness

lemma soundness-strat-helper:
(prov-sdia D P Q → prov-triple (asn P, com-sdia D, asn Q)) ∧
(prov-row q P Q → prov-triple (asn P, com-cell (fst q), asn Q)) ∧
(prov-cell γ P Q → prov-triple (asn P, com-cell γ, asn Q))
(proof)

corollary soundness-strat:
assumes prov-sdia D P Q
shows prov-triple (asn P, com-sdia D, asn Q)
(proof)

end

8 Syntax and proof rules for graphical diagrams

theory Ribbons-Graphical imports
Ribbons-Interfaces
begin

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

8.1 Syntax of graphical diagrams

Fix a type for node identifiers

typedecl node

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

```plaintext
datatype assertion-gadget =
  Rib assertion
| Exists-dia string diagram
and command-gadget =
  Com command
| Choose-dia diagram diagram
| Loop-dia diagram
and diagram = Graph
  node fset
  node ⇒ assertion-gadget
  (node fset × command-gadget × node fset) list
type-synonym labelling = node ⇒ assertion-gadget
type-synonym edge = node fset × command-gadget × node fset
```

Projecting components from a graph

```plaintext
fun vertices :: diagram ⇒ node fset (- V [1000] 1000)
where (Graph V Λ E) ^V = V

term this (is ^V) = (a test) ^V

fun labelling :: diagram ⇒ labelling (- Λ [1000] 1000)
where (Graph V Λ E) ^Λ = Λ

fun edges :: diagram ⇒ edge list (- E [1000] 1000)
where (Graph V Λ E) ^E = E
```

### 8.2 Well formedness of graphical diagrams

**definition** acyclicity :: edge list ⇒ bool

**where**

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

**definition** linearity :: edge list ⇒ bool

**where**

\[\text{linearity } E \equiv \begin{align*}
\text{distinct } E \land (\forall e \in \text{set } E. \forall f \in \text{set } E. \ e \neq f \rightarrow \\
\text{fst3 } e \cap fset f = \{\} \land \\
\text{thd3 } e \cap fset f = \{\} \}
\end{align*}\]

**lemma** linearityD:

**assumes** linearity E

**shows** distinct E

**and** \[\bigwedge e f. \ [ e \in \text{set } E ; f \in \text{set } E ; e \neq f ] \implies \\
\text{fst3 } e \cap fset f = \{\} \land \\
\text{thd3 } e \cap fset f = \{\}\]

⟨proof⟩
lemma linearityD2:
linearity E \Rightarrow (\forall e f. e \in set E \land f \in set E \land e \neq f \rightarrow
\begin{align*}
\text{fst3 } e & \cap \text{thd3 } e = \{||\} \land \\
\text{fst3 } f & \cap \text{thd3 } f = \{||\}
\end{align*}
)

\langle \text{proof} \rangle

\text{inductive}
\begin{align*}
\text{wf-ass} & \colon \text{assertion-gadget} \Rightarrow \text{bool and} \\
\text{wf-com} & \colon \text{command-gadget} \Rightarrow \text{bool and} \\
\text{wf-dia} & \colon \text{diagram} \Rightarrow \text{bool}
\end{align*}
where
\begin{align*}
\text{wf-rib} & \colon \text{wf-ass} (\text{Rib } p) \\
\text{wf-exists} & \colon \text{wf-dia } G \Rightarrow \text{wf-ass} (\text{Exists-dia } x \ G) \\
\text{wf-com} & \colon \text{wf-com} (\text{Com } c) \\
\text{wf-choice} & \colon [\text{wf-dia } G ; \text{wf-dia } H] \Rightarrow \text{wf-com} (\text{Choose-dia } G \ H) \\
\text{wf-loop} & \colon \text{wf-dia } G \Rightarrow \text{wf-com} (\text{Loop-dia } G) \\
\text{wf-dia} & \colon [\forall e \in 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 set E. \text{fst3 } e \cup \text{thd3 } e \subseteq V ] \Rightarrow \text{wf-dia } (\text{Graph } V \Lambda E)
\end{align*}

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

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

\text{8.3 Initial and terminal nodes}

definition
\begin{align*}
\text{initials} & \colon \text{diagram} \Rightarrow \text{node } \text{fset} \\
\text{where} & \\
\text{initials } G = \text{ffilter } (\lambda v. (\forall e \in set G \ast E. v \not\in \text{thd3 } e)) \ G \ast V
\end{align*}

\text{definition}
\begin{align*}
\text{terminals} & \colon \text{diagram} \Rightarrow \text{node } \text{fset} \\
\text{where} & \\
\text{terminals } G = \text{ffilter } (\lambda v. (\forall e \in set G \ast E. v \not\in \text{fst3 } e)) \ G \ast V
\end{align*}

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

lemma no-edges-imp-all-nodes-terminal:
\begin{align*}
\text{terminals } (\text{Graph } V \Lambda []) = V
\end{align*}
lemma initials-in-vertices:
  initials G \sub G^* V
  (proof)

lemma terminals-in-vertices:
  terminals G \sub G^* V
  (proof)

8.4 Top and bottom interfaces

primrec
  top-ass :: assertion-gadget \Rightarrow interface and
  top-dia :: diagram \Rightarrow interface
where
  top-dia (Graph V \Lambda E) = (\bigotimes v \in initials (Graph 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)

primrec
  bot-ass :: assertion-gadget \Rightarrow interface and
  bot-dia :: diagram \Rightarrow interface
where
  bot-dia (Graph V \Lambda E) = (\bigotimes v \in terminals (Graph 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)

8.5 Proof rules for graphical diagrams

inductive
  prov-dia :: [diagram, interface, interface] \Rightarrow bool and
  prov-com :: [command-gadget, interface, interface] \Rightarrow bool and
  prov-ass :: assertion-gadget \Rightarrow bool
where
  Skip: prov-ass (Rib p)
  | Exists: prov-dia G - - \Rightarrow prov-ass (Exists-dia x G)
  | Basic: prov-triple (asn P, c, asn Q) \Rightarrow prov-com (Com c) P Q
  | Choice: [ prov-dia G P Q ; prov-dia H P Q ]
    \Rightarrow prov-com (Choose-dia G H) P Q
  | Loop: prov-dia G P P \Rightarrow prov-com (Loop-dia G) P P
  | Main: \[ wf-dia G ; \bigwedge v. v \in fset G^* V \Rightarrow prov-ass (G^* \Lambda v) ;
    \bigwedge e. e \in set G^* E \Rightarrow prov-com (snd3 e)
    (\bigotimes v \in fset G^* \Lambda v))
      (\bigotimes v \in thd3 e. bot-ass (G^* \Lambda v))]
    \Rightarrow 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

21
8.6 Extracting commands from diagrams

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

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

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

**Assumptions** \( \pi \in \text{lins} \ G \)

**Shows** (distinct \( \pi \))

\[
\begin{align*}
& \quad \land (\text{set} \ \pi = (\text{fset} \ G \cdot V) <++> (\text{set} \ G \cdot E)) \\
& \quad \land (\forall \ i \ j \ v \ e. \ i < \text{length} \ \pi \land j < \text{length} \ \pi \land \pi!i = \text{Inl} \ v \land \pi!j = \text{Inr} \ e \land v \in| \ \text{fst3} \ e \rightarrow i < j) \\
& \quad \land (\forall \ j \ k \ w \ e. \ j < \text{length} \ \pi \land k < \text{length} \ \pi \land \pi!j = \text{Inr} \ e \land \pi!k = \text{Inl} \ w \land w \in| \ \text{thd3} \ e \rightarrow j < k) 
\end{align*}
\]

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

\[
(\forall \ i < \text{length} \ \pi. \ \text{case-sum} (\text{coms-ass} \circ \Lambda) (\text{coms-com} \circ \text{snd3}) (\pi!i) (\text{cs}!i)) =
\]

\[
\begin{align*}
& (\forall \ i. \ i < \text{length} \ \pi \land (\exists \ v. \ (\pi!i) = \text{Inl} \ v) \rightarrow \text{coms-ass} (\Lambda (\text{projl} (\pi!i))) (\text{cs}!i)) \land \\
& (\forall \ i. \ i < \text{length} \ \pi \land (\exists \ e. \ (\pi!i) = \text{Inr} \ e) \rightarrow \text{coms-com} (\text{snd3} (\text{projr} (\pi!i))) (\text{cs}!i)))
\end{align*}
\]

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

**inductive**

\[
\begin{align*}
\text{coms-dia} &:: [\text{diagram}, \text{command}] \Rightarrow \text{bool} \quad \text{and} \\
\text{coms-ass} &:: [\text{assertion-gadget}, \text{command}] \Rightarrow \text{bool} \quad \text{and} \\
\text{coms-com} &:: [\text{command-gadget}, \text{command}] \Rightarrow \text{bool} \\
\end{align*}
\]

**where**

\[
\begin{align*}
\text{coms-skip}: & \quad \text{coms-ass} (\text{Rib } p) \text{ Skip} \\
\text{coms-exists}: & \quad \text{coms-dia } G \ c \Rightarrow \text{coms-ass} (\text{Exists-dia } x \ G) \ c \\
\text{coms-basic}: & \quad \text{coms-com} (\text{Com } c) \ c \\
\text{coms-choice}: & \quad [ \text{coms-dia } G \ c; \text{coms-dia } H \ d ] \Rightarrow \text{coms-com} (\text{Choose-dia } G \ H) \ (\text{Choose } c \ d) \\
\text{coms-loop}: & \quad \text{coms-dia } G \ c \Rightarrow \text{coms-com} (\text{Loop-dia } G) \ (\text{Loop } c) \\
\text{coms-main}: & \quad \forall i < \text{length } \pi. \text{case-sum} (\text{coms-ass } \circ \Lambda) (\text{coms-com } \circ \text{snd3}) (\pi!i) (\text{cs!i}) \\
& \Rightarrow \text{coms-dia} (\text{Graph } V \ \Lambda \ E) \ (\text{foldr} (\circ ;;) \text{ csSkip}) \\
\end{align*}
\]

**monos**

\[
\begin{align*}
\text{coms-mono-helper} \\
\end{align*}
\]

**inductive-cases**

\[
\begin{align*}
\text{coms-skip-inv}: & \quad \text{coms-ass} (\text{Rib } p) \ c \\
\text{coms-exists-inv}: & \quad \text{coms-ass} (\text{Exists-dia } x \ G) \ c \\
\text{coms-basic-inv}: & \quad \text{coms-com} (\text{Com } c') \ c \\
\text{coms-choice-inv}: & \quad \text{coms-com} (\text{Choose-dia } G \ H) \ c \\
\text{coms-loop-inv}: & \quad \text{coms-com} (\text{Loop-dia } G) \ c \\
\text{coms-main-inv}: & \quad \text{coms-dia } G \ c \\
\end{align*}
\]

**end**

### 9 Soundness for graphical diagrams

**theory** Ribbons-Graphical-Soundness

**imports**

Ribbons-Graphical

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 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 ↦ topbot

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

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

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

fun next-ps :: proofstate ⇒ node + edge ⇒ proofstate
where
   next-ps σ (Inl v) = σ ⊖ {{v}} ⊕ [{{v}} |⇒ Bot]
   | next-ps σ (Inr e) = σ ⊖ fst3 e ⊕ [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))

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 op #.

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

Distributing mk-ps-chain over snoc.

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

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

\text{lemma} \text{mk-ps-chain-ccons:}
\begin{align*}
\text{fixes } & \pi \Pi \\
\text{shows } & \text{mk-ps-chain } (\{ |\sigma| \} \cdot x \cdot \Pi) \pi = \{ |\sigma| \} \cdot x \cdot \text{mk-ps-chain } \Pi \pi \\
\end{align*}

\text{⟨proof⟩}

\text{lemma} \text{pre-mk-ps-chain:}
\begin{align*}
\text{fixes } & \Pi \pi \\
\text{shows } & \text{pre } (\text{mk-ps-chain } \Pi \pi) = \text{pre } \Pi \\
\end{align*}

\text{⟨proof⟩}

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.

\text{lemma} \text{comlist-mk-ps-chain:}
\begin{align*}
\text{comlist } (\text{mk-ps-chain } \Pi \pi) = \text{comlist } \Pi \oplus \pi \\
\end{align*}

\text{⟨proof⟩}

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

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

We now give an updated version of the \text{lins } G\ function. This was originally defined in \text{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.

\text{definition} \text{lins2 } :: [\text{node fset, diagram}] \Rightarrow \text{lin set}
\begin{align*}
\text{where}
\text{lins2 } S G \equiv \{ \pi :: \text{lin}. \\
\text{(distinct } \pi) \\
\land (\text{set } \pi = (\text{fset } G' V - \text{fset } S) <+> \text{set } G' E) \\
\land (\forall i j v e. i < \text{length } \pi \land j < \text{length } \pi \\
\land \pi!i = \text{Inl } v \land \pi!j = \text{Inr } e \land v |\in| \text{fst3 } e \longrightarrow 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 \longrightarrow j < k) \}
\end{align*}

25
lemma lins2D:
assumes \( \pi \in \text{lins2 } S G \)
shows distinct \( \pi \)

and set \( \pi = (\text{fset } G^V - \text{fset } S) \leftrightarrow \text{set } G^E \)

and \( \wedge i \ j \ v \ e. \ [i < \text{length } \pi ; j < \text{length } \pi ; \pi!i = \text{Inl } v; \pi!j = \text{Inr } e; v \in \text{fst3 } e] \Rightarrow i < j \)

and \( \wedge j \ k \ w \ e. \ [j < \text{length } \pi ; k < \text{length } \pi ; \pi!j = \text{Inr } e; \pi!k = \text{Inl } w; w \in \text{thd3 } e] \Rightarrow j < k \)

⟨proof⟩

lemma lins2I:
assumes distinct \( \pi \)

and \( \pi = (\text{fset } G^V - \text{fset } S) \leftrightarrow \text{set } G^E \)

and \( \wedge i \ j \ v \ e. \ [i < \text{length } \pi ; j < \text{length } \pi ; \pi!i = \text{Inl } v; \pi!j = \text{Inr } e; v \in \text{fst3 } e] \Rightarrow i < j \)

and \( \wedge j \ k \ w \ e. \ [j < \text{length } \pi ; k < \text{length } \pi ; \pi!j = \text{Inr } e; \pi!k = \text{Inl } w; w \in \text{thd3 } e] \Rightarrow j < k \)

shows \( \pi \in \text{lins2 } S G \)

⟨proof⟩

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

lemma lins-is-lins2-with-empty-S:
\( \text{lins } G = \text{lins2 } \{||\} G \)

⟨proof⟩

The first proofstate for a diagram \( G \) is obtained by mapping each of its initial nodes to \( \text{Top} \).

definition initial-ps :: diagram \Rightarrow proofstate

where

\( \text{initial-ps } G \equiv [ \text{initials } G |=> \text{Top} ] \)

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

definition initial-ps2 :: [node fset, diagram] \Rightarrow proofstate

where

\( \text{initial-ps2 } S G \equiv [ \text{initials } G - S |=> \text{Top} ] \oplus [ S |=> \text{Bot} ] \)

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

lemma initial-ps-is-initial-ps2-with-empty-S:
\( \text{initial-ps = initial-ps2 } \{||\} \)

⟨proof⟩

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

definition ps-chains :: diagram \Rightarrow ps-chain set
where
\[
\text{ps-chains } G \equiv \text{mk-ps-chain } (\text{cNil } (\text{initial-ps } G)) \triangleright \text{lins } G
\]
The following function extracts the set of proofstate chains from a partially-processed diagram. Nodes in \( S \) are excluded from the resulting chains.

**definition**

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

**where**

\[
\text{ps-chains2 } S \ G \equiv \text{mk-ps-chain } (\text{cNil } (\text{initial-ps2 } S \ G)) \triangleright \text{lins2 } S \ G
\]

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

**lemma** \( \text{ps-chains-is-ps-chains2-with-empty-S} \):

\[
\text{ps-chains} = \text{ps-chains2 } \{||\}
\]

⟨proof⟩

We now wish to describe proofstates chain that are well-formed. First, let us say that \( f \oplus \text{disjoint } g \) is defined, when \( f \) and \( g \) have disjoint domains, as \( f \oplus g \). Then, a well-formed proofstate chain consists of triples of the form \( (\sigma \oplus \text{disjoint } [\{v\}] \to Top, \text{Inl } v, \sigma \oplus \text{disjoint } [\{v\}] \to Bot) \), where \( v \) is a node, or of the form \( (\sigma \oplus \text{disjoint } [\{v|s|} \to Bot), \text{Inr } e, \sigma \oplus \text{disjoint } [\{w|s|} \to Top) \), where \( e \) is an edge with source and target nodes \( v|s| \) and \( w|s| \) respectively.

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

**definition**

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

**where**

\[
\text{wf-ps-triple } T = (\text{case snd3 } T \text{ of})
\]

\[
\text{Inl } v \Rightarrow (\exists \sigma. \text{} v \notin fdom \sigma \\
\text{\quad } \land \text{fst3 } T = [\{v\}] \to Top \oplus \sigma \\
\text{\quad } \land \text{thd3 } T = [\{v\}] \to Bot \oplus \sigma)
\]

\[
\text{Inr } e \Rightarrow (\exists \sigma. (\text{fst3 } e \cup thd3 e) \cap fdom \sigma = \{||\}) \\
\text{\quad } \land \text{fst3 } T = [\text{fst3 } e \to Bot \oplus \sigma] \\
\text{\quad } \land \text{thd3 } T = [\text{thd3 } e \to Top \oplus \sigma])
\]

**lemma** \( \text{wf-ps-triple-nodeI} \):

**assumes** \( \exists \sigma. \text{} v \notin fdom \sigma \land \sigma 1 = [\{v\}] \to Top \oplus \sigma \land \sigma 2 = [\{v\}] \to Bot \oplus \sigma \)

**shows** \( \text{wf-ps-triple } (\sigma 1, \text{Inl } v, \sigma 2) \)

⟨proof⟩

**lemma** \( \text{wf-ps-triple-edgeI} \):

**assumes** \( \exists \sigma. (\text{fst3 } e \cup thd3 e) \cap fdom \sigma = \{||\}) \\
\text{\quad } \land \sigma 1 = [\text{fst3 } e \to Bot \oplus \sigma] \\
\text{\quad } \land \sigma 2 = [\text{thd3 } e \to Top \oplus \sigma) \)

**shows** \( \text{wf-ps-triple } (\sigma 1, \text{Inr } e, \sigma 2) \)
\textbf{definition}

\texttt{wf-ps-chain :: \texttt{ps-chain} ⇒ bool}

\texttt{where}

\texttt{wf-ps-chain ≡ chain-all \texttt{wf-ps-triple}}

\textbf{lemma} \texttt{next-initial-ps2-vertex:}

\texttt{initial-ps2 (\{\|v\|\} \cup \texttt{S}) \texttt{G}}

\texttt{= initial-ps2 \texttt{S \texttt{G} \cup \{\|v\|\} \cup \{\|v\|\} \Rightarrow Bot}}

\texttt{(proof)}

\textbf{lemma} \texttt{next-initial-ps2-edge:}

\texttt{assumes \texttt{G} = \texttt{Graph V \Lambda E and G'} = \texttt{Graph V' \Lambda E' and}}

\texttt{\texttt{V'} = \texttt{V - fst3 e and E'} = removeAll e E and e \in \texttt{set E and}}

\texttt{\texttt{fst3 e \subseteq} \texttt{S and S \subseteq initialize} \texttt{G and \texttt{wf-dia} \texttt{G}}}

\texttt{shows \texttt{initial-ps2 (S - fst3 e) G'}}

\texttt{= initial-ps2 \texttt{S G} \cup \texttt{fst3 e} \cup \{\texttt{thd3 e} \Rightarrow \texttt{Top}\}}

\texttt{(proof)}

\textbf{lemma} \texttt{next-lins2-vertex:}

\texttt{assumes \texttt{Inl v \# \pi \in \texttt{lins2 S G}}}

\texttt{assumes \texttt{v \notin S}}

\texttt{shows \pi \in \texttt{lins2 (\{\|v\|\} \cup \texttt{S}) G}}

\texttt{(proof)}

\textbf{lemma} \texttt{next-lins2-edge:}

\texttt{assumes \texttt{Inr e \# \pi \in \texttt{lins2 S (Graph V \Lambda E)}}}

\texttt{and \texttt{vs \subseteq S}}

\texttt{and \texttt{e = (vs, c, ws)}}

\texttt{shows \pi \in \texttt{lins2 (S - vs) (Graph (V - vs) \Lambda (removeAll e E))}}

\texttt{(proof)}

We wish to prove that every proofstate chain that can be obtained from a linear extension of \texttt{G} is well-formed and has as its final proofstate that state in which every terminal node in \texttt{G} is mapped to \texttt{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 \texttt{(G, S)} is defined as the number of nodes in \texttt{G}, plus the number of edges, minus the number of nodes in \texttt{S}.

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

\textbf{lemma} \texttt{wf-chains2:}

\texttt{fixes k}

\texttt{assumes \texttt{S \subseteq initial} \texttt{G}}

\texttt{and \texttt{wf-dia G}}

\texttt{and \Pi \in \texttt{ps-chains2 S G}}

28
\[ \text{and } \text{fcard } G \cdot V + \text{length } G \cdot E = k + \text{fcard } S \]
\[ \text{shows } \text{wf-ps-chain } \Pi \land (\text{post } \Pi = [ \text{terminals } G \Rightarrow \text{Bot }]) \]
(\text{proof})

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

9.2 Interface chains

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

An interface chain is similar to a proofstate chain. However, where a proof-state 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.

\begin{itemize}
\item \text{definition } \text{ps-to-int } :: [\text{diagram, proofstate}] \Rightarrow \text{interface}
\text{where}
\text{ps-to-int } G \sigma \equiv \bigotimes v \mid \in | \text{fdom } \sigma. \text{case-topbot top-ass bot-ass (lookup } \sigma \ v) \ (G \Lambda v)
\end{itemize}

\begin{itemize}
\item \text{definition } \text{ps-chain-to-int-chain } :: [\text{diagram, ps-chain}] \Rightarrow \text{int-chain}
\text{where}
\text{ps-chain-to-int-chain } G \Pi \equiv \text{chainmap (ps-to-int } G ((\text{case-sum } (\text{Inl } \circ \ G \Lambda) (\text{Inr } \circ \ \text{snd3})))) \Pi
\end{itemize}

\begin{itemize}
\item \text{lemma } \text{ps-chain-to-int-chain-simp:}
\text{ps-chain-to-int-chain } (\text{Graph } V \Lambda E) \Pi = \text{chainmap (ps-to-int } (\text{Graph } V \Lambda E)) ((\text{case-sum } (\text{Inl } \circ \ \Lambda) (\text{Inr } \circ \ \text{snd3})))) \Pi
\end{itemize}

(\text{proof})

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

** Shows**

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

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

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

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

** Assumes** prov-dia $G P Q$

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

*Proof*

**End**

### References


