Ribbon Proofs for Separation Logic
(Isabelle Formalisation)

John Wickerson

September 19, 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
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
\[ \text{map-diff} :: (\forall k \Rightarrow (\forall v) \Rightarrow (\forall k \Rightarrow (\forall v)) \Rightarrow (\forall k \Rightarrow (\forall v)) \]
where
\[ \text{map-diff} f ks = \text{restrict-map} f (- fset ks) \]

lift-definition
\[ \text{fmap-diff} :: (\forall k \Rightarrow (\forall v) \Rightarrow (\forall k \Rightarrow (\forall v)) \Rightarrow (\forall k \Rightarrow (\forall v)) \]
is \text{map-diff}
⟨proof⟩

abbreviation
\[ \text{FMAP-DIFF} :: (\forall k \Rightarrow (\forall v) \Rightarrow (\forall k \Rightarrow (\forall v)) \text{ (infix \ o 110)} \]
where
\( xs \ominus ys \equiv \text{fmap-diff} xs ys \)

2.3 Comprehension

definition
\[ \text{make-map} :: (\forall k \Rightarrow (\forall v) \Rightarrow (\forall k \Rightarrow (\forall v)) \]
where
\[ \text{make-map} ks v = \lambda k. \text{if} k \in fset ks \text{ then} Some v \text{ else} None \]

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

lift-definition
\[ \text{make-fmap} :: (\forall k \Rightarrow (\forall v) \Rightarrow (\forall k \Rightarrow (\forall v)) \]
is \text{make-map}
⟨proof⟩

abbreviation
\[ \text{MAKE-FMAP} :: (\forall k \Rightarrow (\forall v) \Rightarrow (\forall k \Rightarrow (\forall v)) \text{ ([ - |\Rightarrow - ])} \]
where
\[ \text{[ ks |\Rightarrow v ]} \equiv \text{make-fmap} ks v \]

2.4 The empty finite partial function

lift-definition
\[ \text{fmap-empty} :: (\forall k \Rightarrow (\forall v) \]
is \text{empty}
⟨proof⟩

2.5 Domain

definition
\[ \text{dom-fset} :: (\forall k \Rightarrow (\forall v) \Rightarrow (\forall k \Rightarrow (\forall v)) \]
where \( \text{dom-fset} f \equiv \text{THE} x. \text{fset} x = \text{dom} f \)
lift-definition
  fdom :: ('k → 'v) ⇒ 'k fset
is dom-fset ⟨proof⟩

lemma inv-fset:
  assumes finite X
  shows ∃!x. fset x = X
⟨proof⟩

lemma fset-inv-fset:
  assumes finite X
  shows fset (THE x. fset x = X) = X
⟨proof⟩

lemma fset-dom-fset:
  assumes finite (dom f)
  shows fset (dom-fset f) = dom f
⟨proof⟩

lemma fmap-add-commute:
  assumes fdom A ⋂ fdom B = {||}
  shows A ⊕ B = B ⊕ A
⟨proof⟩

lemma make-fmap-union:
  [ xs |⇒> v ] ⊕ [ ys |⇒> v ] = [ xs |∪| ys |⇒> v ]
⟨proof⟩

lemma fdom-union:
  fdom (xs ⊕ ys) = fdom xs |∪| fdom ys
⟨proof⟩

lemma fdom-make-fmap:
  fdom [ ks |⇒> v ] = ks
⟨proof⟩

2.6 Lookup

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

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

lemma lookup-make-fmap1:
lemma lookup-union1:
  assumes k ∈ fdom ys
  shows lookup (xs ⊕ ys) k = lookup ys k
⟨proof⟩
lemma lookup-union2:
  assumes k ∉ fdom ys
  shows lookup (xs ⊕ ys) k = lookup xs k
⟨proof⟩
lemma lookup-union3:
  assumes k ∉ fdom xs
  shows lookup (xs ⊕ ys) k = lookup ys k
⟨proof⟩
end

3 General purpose definitions and lemmas

theory JHelper imports Main begin

lemma Collect-iff:
  a ∈ {x . P x} ≡ P a
⟨proof⟩
lemma diff-diff-eq:
  assumes C ⊆ B
  shows (A − C) − (B − C) = A − B
⟨proof⟩
lemma nth-in-set:
  [ i < length xs ; xs ! i = x ] ⇒ x ∈ set xs ⟨proof⟩
lemma disjI [intro]:
  assumes ¬ P ⇒ Q
  shows P ∨ Q
⟨proof⟩
lemma empty-eq-Plus-conv:
  (∅ = A <+> B) = (A = ∅ ∧ B = ∅)
⟨proof⟩
3.1 Projection functions on triples

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

**where** $\text{fst3} \equiv \text{fst}$

**definition** $\text{snd3} :: 'a \times 'b \times 'c \Rightarrow 'b$

**where** $\text{snd3} \equiv \text{fst} \circ \text{snd}$

**definition** $\text{thd3} :: 'a \times 'b \times 'c \Rightarrow 'c$

**where** $\text{thd3} \equiv \text{snd} \circ \text{snd}$

**lemma** $\text{fst3-simp}$:
\[
\forall a \ b \ c. \ \text{fst3} \ (a,b,c) = a
\]

**proof**

**lemma** $\text{snd3-simp}$:
\[
\forall a \ b \ c. \ \text{snd3} \ (a,b,c) = b
\]

**proof**

**lemma** $\text{thd3-simp}$:
\[
\forall a \ b \ c. \ \text{thd3} \ (a,b,c) = c
\]

**proof**

**lemma** $\text{tripleI}$:

**fixes** $T \ \text{U}$

**assumes** $\text{fst3} \ T = \text{fst3} \ \text{U}$

**and** $\text{snd3} \ T = \text{snd3} \ \text{U}$

**and** $\text{thd3} \ T = \text{thd3} \ \text{U}$

**shows** $T = \text{U}$

**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 $\text{SMain}$ proof rule, and for conducting the proof of soundness.

**datatype** $(a,c) \ \text{chain} =$
\[
\begin{align*}
\text{cNil} \ a & \quad (\emptyset - \emptyset) \\
\mid \ \text{cCons} \ a \ 'c \ (a,c) \ \text{chain} & \quad (\emptyset - \cdot \cdot \cdot [0,0,0] \ 60)
\end{align*}
\]

For example, $\emptyset \ a \cdot \text{proof} \cdot \emptyset \ \text{chain} \cdot \text{might} \cdot \emptyset \ \text{look} \cdot \cdot \cdot \ \text{like} \cdot \emptyset \ \text{this} \}$.
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 (\{ - \} · Π) = post Π

Project list of commands.

fun
comlist :: ('a,'c) chain ⇒ 'c list
where
comlist \{ - \} = []
| comlist (\{ - \} · x · Π) = x # (comlist Π)

4.2 Chain length

fun
chainlen :: ('a,'c) chain ⇒ nat
where
chainlen \{ - \} = 0
| chainlen (\{ - \} · · · Π) = 1 + (chainlen Π)

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

4.3 Extracting triples from chains

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

fun
nthtriple :: ('a,'c) chain ⇒ nat ⇒ ('a * 'c * 'a)
where
nthtriple (\{ P \} · x · Π) 0 = (P, x, pre Π)
| nthtriple (\{ P \} · x · Π) (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 \( i < \text{chainlen } \Pi \implies \text{snd3 } (\text{nthtriple } \Pi \ i) = (\text{comlist } \Pi)!i \)

\langle proof \rangle

### 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**

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

**where**

\[ \text{chain-all } \varphi \ [\sigma] = \text{True} \]

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

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

\[ x \leq y \implies \text{chain-all } x \leq \text{chain-all } y \]

\langle proof \rangle

**lemma** \( \text{chain-all-nthtriple}: \)

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

\langle proof \rangle

### 4.5 A map function for proof chains

\( \text{chainmap } f \ g \ Pi \) maps \( f \) over each of \( Pi \)'s assertions, and \( g \) over each of \( Pi \)'s commands.

**fun**

\[ \text{chainmap } :: (\text{'a } \Rightarrow \text{'c }) \Rightarrow (\text{'c } \Rightarrow \text{'d} ) \Rightarrow \text{('a,'c) chain } \Rightarrow \text{('b,'d) chain} \]

**where**

\[ \text{chainmap } f \ g \ [\ P \] = [\ f \ P \]

\[ \text{chainmap } f \ g \ (\ [\ P \] \cdot x \cdot \Pi) = [\ f \ P \] \cdot g \ x \cdot \text{chainmap } f \ g \Pi \]

Mapping over a chain preserves its length.

**lemma** \( \text{chainmap-preserves-length}: \)

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

\langle proof \rangle

**lemma** \( \text{pre-chainmap}: \)

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

\langle proof \rangle

**lemma** \( \text{post-chainmap}: \)

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

\langle proof \rangle

**lemma** \( \text{nthtriple-chainmap}: \)

**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 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 · II) y τ = {σ} · x · (cSnoc y τ)

lemma len-snoc:
  fixes II x P
  shows chainlen (cSnoc II x P) = 1 + (chainlen II)
⟨proof⟩

lemma pre-snoc:
  pre (cSnoc II x P) = pre II
⟨proof⟩

lemma post-snoc:
  post (cSnoc II x P) = P
⟨proof⟩

lemma comlist-snoc:
  comlist (cSnoc II x p) = comlist II @ [x]
⟨proof⟩

end

5 Assertions, commands, and separation logic proof rules

theory Ribbons-Basic imports  Main
begin

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

5.1 Assertions

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

typedecl assertion

axiomatization
Emp :: assertion

axiomatization
Star :: assertion ⇒ assertion ⇒ assertion (infixr ⋆ 55)

where
star-comm: p ⋆ q = q ⋆ p and
star-assoc: (p ⋆ q) ⋆ r = p ⋆ (q ⋆ r) and
star-emp: p ⋆ Emp = p and
emp-star: Emp ⋆ p = p

lemma star-rot:
q ⋆ p ⋆ r = p ⋆ q ⋆ r
⟨proof⟩

axiomatization
Exists :: string ⇒ assertion ⇒ assertion

Extracting the set of program variables mentioned in an assertion.

axiomatization
rd-ass :: assertion ⇒ string set

where
rd-emp: rd-ass Emp = {}
and rd-star: rd-ass (p ⋆ q) = rd-ass p ∪ rd-ass q
and rd-exists: rd-ass (Exists x p) = rd-ass p

5.2 Commands

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

typedecl command

axiomatization
Choose :: command ⇒ command ⇒ command

axiomatization
Loop :: command ⇒ command

axiomatization
Skip :: command

axiomatization
Seq :: command ⇒ command ⇒ command (infixr ;; 55)

where
seq-assoc: c1 ;; (c2 ;; c3) = (c1 ;; c2) ;; c3
and seq-skip: c ;; Skip = c
and skip-seq: Skip ;; c = c

Extracting the set of program variables read by a command.

axiomatization
rd-com :: command ⇒ string set
where \( \text{rd-com-choose}: \text{rd-com} \ (\text{Choose} \ c_1 \ c_2) = \text{rd-com} \ c_1 \cup \text{rd-com} \ c_2 \)
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 \( \text{rd-com-seq}: \text{rd-com} \ (c_1 ;; c_2) = \text{rd-com} \ c_1 \cup \text{rd-com} \ c_2 \)

Extracting the set of program variables written by a command.

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

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
\( \text{prov-triple} :: \text{assertion} \times \text{command} \times \text{assertion} \Rightarrow \text{bool} \)
where
\( \exists x: \text{prov-triple} \ (p, c, q) = \text{prov-triple} \ (\exists x \ p, c, \exists x \ q) \)
\( \Rightarrow \text{prov-triple} \ (p, \text{Choose} c_1 c_2, q) \)
\( \Rightarrow \text{prov-triple} \ (p, \text{Loop} c, p) \)
\( \Rightarrow \text{prov-triple} \ (p, \text{Frame} \ c, p) \)
\( \Rightarrow \text{prov-triple} \ (p, \text{Skip} \ p) \)
\( \Rightarrow \text{prov-triple} \ (p, \text{Seq} \ (c_1 ;; c_2, q, r)) \)

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

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

lemma loop-lemma:
\( \text{assumes } \text{prov-triple} \ (p_1, c, q_1) \text{ and } p = p_1 \text{ and } p_1 = q_1 \text{ and } q_1 = q \)
\( \text{shows } \text{prov-triple} \ (p, \text{Loop} c, q) \) (proof)

lemma seq-lemma:
\( \text{assumes } \text{prov-triple} \ (p_1, c_1, q_1) \text{ and } \text{prov-triple} \ (p_2, c_2, q_2) \)
and \( q_1 = p_2 \)
\( \text{shows } \text{prov-triple} \ (p_1, c_1 ;; c_2, q_2) \)
⟨proof⟩

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 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 ⊗_c 50)
  ∣ Emp-int-conc (ε_c)
  ∣ Exists-int-conc string conc-interface

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

inductive equiv-int :: conc-interface ⇒ conc-interface ⇒ bool (infix≃ 45)
where
  refl: P≃P
  | sym: P≃Q ⇒ Q≃P
  | trans: [[P≃Q; Q≃R]] ⇒ P≃R
  | cong-hcomp1: P≃Q ⇒ P′⊗_c P≃P′⊗_c Q
  | cong-hcomp2: P≃Q ⇒ P⊗_c P′≃Q⊗_c P′
  | cong-exists: P≃Q ⇒ Exists-int-conc x P≃Exists-int-conc x Q
  | hcomp-conc-assoc: P⊗_c (Q⊗_c R)≃(P⊗_c Q)⊗_c R
  | hcomp-conc-comm: P⊗_c Q≃Q⊗_c P
  | hcomp-conc-unit1: ε_c⊗_c P≃P
  | hcomp-conc-unit2: P⊗_c ε_c≃P

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

\textbf{proof}

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

\textbf{lift-definition}

\textit{Ribbon} :: \textit{assertion} \Rightarrow \textit{interface}

is \textit{Ribbon-conc} \textbf{proof}

\textbf{lift-definition}

\textit{Emp-int} :: \textit{interface} (\varepsilon)

is \varepsilon_c \textbf{proof}

\textbf{lift-definition}

\textit{Exists-int} :: \textit{string} \Rightarrow \textit{interface} \Rightarrow \textit{interface}

is \textit{Exists-int-conc} \textbf{proof}

\textbf{lift-definition}

\textit{HComp-int} :: \textit{interface} \Rightarrow \textit{interface} \Rightarrow \textit{interface} (\textit{infix} \otimes 50)

is \textit{HComp-int-conc} \textbf{proof}

\textbf{lemma} \textit{hcomp-comm}:

\[(P \otimes Q) = (Q \otimes P)\]

\textbf{proof}

\textbf{lemma} \textit{hcomp-assoc}:

\[(P \otimes (Q \otimes R)) = ((P \otimes Q) \otimes R)\]

\textbf{proof}

\textbf{lemma} \textit{emp-hcomp}:

\[\varepsilon \otimes P = P\]

\textbf{proof}

\textbf{lemma} \textit{hcomp-emp}:

\[P \otimes \varepsilon = P\]

\textbf{proof}

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

\textbf{proof}

\textbf{6.2 An iterated horizontal-composition operator}

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

\textbf{where}

\textit{iter-hcomp} \textit{X f} \equiv \textit{ffold} (\textit{op} \otimes \textit{f}) \varepsilon X
syntax iter-hcomp-syntax ::
  'a ⇒ ('a fset) ⇒ ('a ⇒ interface) ⇒ interface
translations \( \otimes x \in |M. \; e = \mathop{\text{CONST iter-hcomp}} (\lambda x. \; e) \)

term \( \otimes P \in |Ps. \; f \) — this is eta-expanded, so prints in expanded form
term \( \otimes P \in |Ps. \; f \) — this isn’t eta-expanded, so prints as written

lemma iter-hcomp-cong:
  assumes \( \forall v \in fset \mathbin{\text{vs}}. \; \varphi v = \varphi' v \)
  shows \( \otimes v \in |\mathbin{\text{vs}}. \; \varphi v = (\otimes v \in |\mathbin{\text{vs}}. \; \varphi' v) \)
⟨proof⟩

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

lemma iter-hcomp-insert:
  assumes \( v \not\in |\mathbin{\text{us}} \)
  shows \( \otimes x \in |\mathbin{\text{finset}} v \mathbin{\text{vs}}. \; p x) = (p v \otimes (\otimes x \in |\mathbin{\text{us}}. \; p x)) \)
⟨proof⟩

lemma iter-hcomp-union:
  assumes \( \mathbin{\text{vs}} \cap \mathbin{\text{ws}} = {||} \)
  shows \( \otimes x \in |\mathbin{\text{vs}} \cup \mathbin{\text{ws}}. \; p x) = ((\otimes x \in |\mathbin{\text{vs}}. \; p x) \otimes (\otimes x \in |\mathbin{\text{ws}}. \; p x)) \)
⟨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 \otimes_c Q) = (conc-asn P) \* (conc-asn Q)
| conc-asn (\varepsilon_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 \otimes Q) = (asn P) \* (asn Q)
| asn \varepsilon = Emp
| asn (Exists-int x P) = Exists x (asn P)
6.4 Program variables mentioned in an interface.

fun  
\textit{rd-conc-int} :: \textit{conc-interface} \Rightarrow \textit{string set}

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

lift-definition
\textit{rd-int} :: \textit{interface} \Rightarrow \textit{string set}
is \textit{rd-conc-int}

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

lemma \textit{rd-int-is-rd-ass}:
\textit{rd-ass } (\textit{asn } P) = \textit{rd-int } P

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

lemma \textit{seq-fold}:
\begin{align*}
&\forall \Pi. \{ \text{length } cs = \text{chainlen } \Pi ; p1 = \textit{asn } (\text{pre } \Pi) ; p2 = \textit{asn } (\text{post } \Pi) ; \\
&\forall i. \text{ } (\text{asn } (\text{fst3 } (\text{nthtriple } \Pi i)), \text{ } cs ! \text{ } i, \text{ } \textit{asn } (\text{thd3 } (\text{nthtriple } \Pi i))) \} \\
&\implies \text{prov-triple } (p1, \text{foldr } (\text{op };; ) \text{ } cs \text{ } \text{Skip}, \text{ } p2)
\end{align*}

7 Syntax and proof rules for stratified diagrams

theory \textit{Ribbons-Stratified} imports
\textit{Ribbons-Interfaces}  
\textit{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 \textit{sdiagram} = \textit{SDiagram} (\text{cell} × \textit{interface}) \text{ list}

and \text{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
  com-sdia :: sdiagram ⇒ command and
  com-cell :: cell ⇒ command
where
  com-sdia (SDiagram ϱ s) = foldr (op ;;) (map (com-cell ◦ fst) ϱ s) Skip
| com-cell (Filler P) = Skip
| com-cell (Basic P c Q) = c
| com-cell (Exists-sdia x D) = com-sdia D
| com-cell (Choose-sdia P D E Q) = Choose (com-sdia D) (com-sdia E)
| com-cell (Loop-sdia P D Q) = Loop (com-sdia D)

Extracting the program variables written by a stratified diagram.

fun
  wr-sdia :: sdiagram ⇒ string set and
  wr-cell :: cell ⇒ string set
where
  wr-sdia (SDiagram ϱ s) = (∪ r ∈ set ϱ s. 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 gs :: row list
  and g :: row
  shows (wr-sdia D = wr-com (com-sdia D))
  and (wr-cell γ = wr-com (com-cell γ))
  and (∪ g ∈ set gs. wr-cell (fst g))
    = wr-com (foldr (op ;;) (map (λ γ. F). com-cell γ) gs) Skip
  and wr-cell (fst g) = wr-com (com-cell (fst g))
⟨proof⟩

7.2 Proof rules for stratified diagrams

inductive
prov-sdia :: [sdiagram, interface, interface] ⇒ bool and
prov-row :: [row, interface, interface] ⇒ bool and
prov-cell :: [cell, interface, interface] ⇒ bool

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

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
typedcl 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
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 (\text{-} `V [1000] 1000)
where (Graph V Λ E) `V = V

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

fun labelling :: diagram ⇒ labelling (\text{-} `Λ [1000] 1000)
where (Graph V Λ E) `Λ = Λ

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

### 8.2 Well formedness of graphical diagrams

```plaintext
definition acyclicity :: edge list ⇒ bool
where
  acyclicity E ≡ acyclic (\bigcup e ∈ set E. fset (fst3 e) × fset (thd3 e))

definition linearity :: edge list ⇒ bool
where
  linearity E ≡
  distinct E ∧ (∀ e ∈ set E. ∀ f ∈ set E. e ≠ f −→
    fst3 e \cap| fst3 f = {||} ∧
    thd3 e \cap| thd3 f = {||})

lemma linearityD:
  assumes linearity E
  shows distinct E
  and \bigwedge e f. [ e ∈ set E ; f ∈ set E ; e ≠ f ] −→
  fst3 e \cap| fst3 f = {||} ∧
  thd3 e \cap| thd3 f = {||}
⟨proof⟩
```

19
**Lemma** *linearityD2:*

\[\text{linearity } E \implies (\forall e, f. \; e \in \text{set } E \land f \in \text{set } E \land e \neq f \implies \]
\[\text{fst3 } e \cap \text{fst3 } f = \{\|\} \land \]
\[\text{thd3 } e \cap \text{thd3 } f = \{\|\}\]

**Proof**

**Inductive**

\[\text{wf-ass} :: \text{assertion-gadget} \Rightarrow \text{bool and} \]
\[\text{wf-com} :: \text{command-gadget} \Rightarrow \text{bool and} \]
\[\text{wf-dia} :: \text{diagram} \Rightarrow \text{bool} \]

**Where**

\[\text{wf-rib}: \text{wf-ass} (\text{Rib } p) \]
\[\text{wf-exists}: \text{wf-dia } G \implies \text{wf-ass } (\exists \text{ dia } x \; G) \]
\[\text{wf-com}: \text{wf-com } (\text{Com } c) \]
\[\text{wf-choice}: [ \text{wf-dia } G ; \text{wf-dia } H ] \implies \text{wf-com } (\text{Choose-dia } G \; H) \]
\[\text{wf-loop}: \text{wf-dia } G \implies \text{wf-com } (\text{Loop-dia } G) \]
\[\text{wf-dia}: [ \forall e \in \text{set } E. \; \text{wf-com } (\text{snd3 } e) ; \forall v \in \text{fset } V. \; \text{wf-ass } (\Lambda \; v) ; \]
\[\text{acyclicity } E ; \text{linearity } E ; \forall e \in \text{set } E. \; \text{fst3 } e \cup \text{thd3 } e \subseteq V ] \implies \]
\[\text{wf-dia } (\text{Graph } V \; \Lambda \; E) \]

**Inductive-cases** *wf-dia-inv':*  
\[\text{wf-dia } (\text{Graph } V \; \Lambda \; E)\]

**Lemma** *wf-dia-inv:*

\[\text{assumes} \; \text{wf-dia } (\text{Graph } V \; \Lambda \; E) \]
\[\text{shows} \; \forall v \in \text{fset } V. \; \text{wf-ass } (\Lambda \; v) \]
\[\text{and} \; \forall e \in \text{set } E. \; \text{wf-com } (\text{snd3 } e) \]
\[\text{and} \; \text{acyclicity } E \]
\[\text{and} \; \text{linearity } E \]
\[\text{and} \; \forall e \in \text{set } E. \; \text{fst3 } e \cup \text{thd3 } e \subseteq V \]

**Proof**

**8.3 Initial and terminal nodes**

**Definition**

\[\text{initials} :: \text{diagram} \Rightarrow \text{node } \text{fset} \]

**Where**

\[\text{initials } G = \text{ffilter } (\lambda v. \; (\forall e \in \text{set } G \cup E. \; v \not\in \text{thd3 } e)) \; G \setminus V \]

**Definition**

\[\text{terminals} :: \text{diagram} \Rightarrow \text{node } \text{fset} \]

**Where**

\[\text{terminals } G = \text{ffilter } (\lambda v. \; (\forall e \in \text{set } G \cup E. \; v \not\in \text{fst3 } e)) \; G \setminus V \]

**Lemma** *no-edges-imp-all-nodes-initial:*

\[\text{initials } (\text{Graph } V \; \Lambda \; []) = V \]

**Proof**

**Lemma** *no-edges-imp-all-nodes-terminal:*

\[\text{terminals } (\text{Graph } V \; \Lambda \; []) = V \]
lemma initials-in-vertices:
  initials G |⊆| G ´V
(\textit{proof})

lemma terminals-in-vertices:
  terminals G |⊆| G ´V
(\textit{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 ; \( \land v. v \in fset G ´V \) ⇒ prov-ass (G ´V) ;
  \( \land e. e \in set G ´E \) ⇒ prov-com (snd3 e)
  (∏ v |∈| fst3 e. bot-ass (G ´V))
  (∏ v |∈| thd3 e. top-ass (G ´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
8.6 Extracting commands from diagrams

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

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

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

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

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

**shows** (distinct \( \pi \))

and (set \( \pi = (\text{fset} \; G \\ V) <+> (\text{set} \; G \;
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 v \; \in \{ | \; \text{fst3} \; e \; \rightarrow \; 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 \; \in \{ | \; \text{thd3} \; e \; \rightarrow \; j < k \})

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

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

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

(\( \forall \; i < \text{length} \; \pi. \; \text{case-sum} \; (\text{coms-ass} \; \circ \; \Lambda) \; (\text{coms-com} \; \circ \; \text{snd}3) \; (\pi!i) \; (cs!i)) \)

\( = \)

((\( \forall \; i < \text{length} \; \pi \land (\exists \; v. \; (\pi!i) = \text{Inl} \; v) \rightarrow \text{coms-ass} \; (\Lambda \; (\text{projl} \; (\pi!i))) \; (cs!i)) \) \land

(\( \forall \; i < \text{length} \; \pi \land (\exists \; e. \; (\pi!i) = \text{Inr} \; e) \rightarrow \text{coms-com} \; (\text{snd}3 \; (\text{projr} \; (\pi!i))) \; (cs!i)) \))

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

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**

- $\text{coms-dia} :: [\text{dia}, \text{cmd}] \Rightarrow \text{bool} \quad \text{and}$
- $\text{coms-ass} :: [\text{assert-gadget}, \text{cmd}] \Rightarrow \text{bool} \quad \text{and}$
- $\text{coms-com} :: [\text{cmd-gadget}, \text{cmd}] \Rightarrow \text{bool}$

**where**

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

**monos**

- $\text{coms-mono-helper}$

**inductive-cases**

- $\text{coms-skip-inv}: \text{coms-ass} (\text{Rib} p) c$
- $\text{coms-exists-inv}: \text{coms-ass} (\text{Exists-dia} x G) c$
- $\text{coms-basic-inv}: \text{coms-com} (\text{Com} c') c$
- $\text{coms-choice-inv}: \text{coms-com} (\text{Choose-dia} G H) c$
- $\text{coms-loop-inv}: \text{coms-com} (\text{Loop-dia} G) c$
- $\text{coms-main-inv}: \text{coms-dia} G c$

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

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

**Lemma mk-ps-chain-ccons:**

- **Fixes** \( \pi \Pi \)
- **Shows** \( \text{mk-ps-chain} (\{\sigma\} \cdot x \cdot \Pi) \pi = \{\sigma\} \cdot x \cdot \text{mk-ps-chain} \Pi \pi \)

**Proof**

**Lemma pre-mk-ps-chain:**

- **Fixes** \( \Pi \pi \)
- **Shows** \( \text{pre} (\text{mk-ps-chain} \Pi \pi) = \text{pre} \Pi \)

**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.

**Lemma comlist-mk-ps-chain:**

- \( \text{comlist} (\text{mk-ps-chain} \Pi \pi) = \text{comlist} \Pi \circ \pi \)

**Proof**

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

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

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

**Definition lins2 :: [node fset, diagram] => lin set**

\[
\text{lins2} \ S \ G \equiv \{ \pi :: \text{lin} . \quad \\
\quad (\text{distinct } \pi) \\
\quad \land \ (\text{set } \pi = (\text{fset } G^V \setminus \text{fset } S) <+> \text{ set } G^E) \\
\quad \land \ (\forall i j \ e. \ i < \text{length } \pi \land j < \text{length } \pi \quad \\
\quad \land \ \pi \! i = \text{Inl } v \land \pi \! j = \text{Inr } e \land v |\in| \text{fst3 } e \rightarrow i < j) \\
\quad \land \ (\forall j k \ w. \ j < \text{length } \pi \land k < \text{length } \pi \quad \\
\quad \land \ \pi \! j = \text{Inr } e \land \pi \! k = \text{Inl } w \land w |\in| \text{thd3 } e \rightarrow j < k) \}
\]
lemma \textit{lins2D}:
\begin{itemize}
\item \textbf{assumes} \( \pi \in \text{lins2} \ S \ G \)
\item \textbf{shows} \( \text{distinct} \ \pi \)
\item \( \text{and} \ \set \ \pi = (\text{fset} \ G^*V - \text{fset} \ S) <+< \text{set} \ G^*E \)
\item \( \text{and} \ \bigwedge i \ j \ v \ e. \ [ \ i < \text{length} \ \pi ; \ j < \text{length} \ \pi \ ; \ 
\pi!i = \text{Inl} \ v ; \ \pi!j = \text{Inr} \ e ; \ v \mid \in \text{fst3} \ e \ ] \Rightarrow \ i < j \)
\item \( \text{and} \ \bigwedge i \ k \ w \ e. \ [ \ j < \text{length} \ \pi ; \ k < \text{length} \ \pi \ ; \ 
\pi!j = \text{Inr} \ e ; \ \pi!k = \text{Inl} \ w ; \ w \mid \in \text{thd3} \ e \ ] \Rightarrow \ j < k \)
\end{itemize}
\langle proof \rangle

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

\begin{itemize}
\item \textbf{lemma} \textit{lins-is-lins2-with-empty-S}:
\item \( \text{lins} \ G = \text{lins2} \ \{||\} \ G \)
\item \langle proof \rangle
\end{itemize}

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

\begin{itemize}
\item \textbf{definition} \textit{initial-ps} :: \text{diagram} \Rightarrow \text{proofstate}
\item \textbf{where} \( \text{initial-ps} \ G \equiv \{ \text{initials} \ G |\Rightarrow \text{Top} \} \)
\end{itemize}

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}.

\begin{itemize}
\item \textbf{definition} \textit{initial-ps2} :: [\text{node fset}, \text{diagram}] \Rightarrow \text{proofstate}
\item \textbf{where} \( \text{initial-ps2} \ S \ G \equiv \{ \text{initials} \ G - S |\Rightarrow \text{Top} \} \oplus \{ S |\Rightarrow \text{Bot} \} \)
\end{itemize}

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

\begin{itemize}
\item \textbf{lemma} \textit{initial-ps-is-initial-ps2-with-empty-S}:
\item \( \text{initial-ps} = \text{initial-ps2} \ \{||\} \)
\item \langle proof \rangle
\end{itemize}

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

\begin{itemize}
\item \textbf{definition} \textit{ps-chains} :: \text{diagram} \Rightarrow \text{ps-chain set}
\end{itemize}

26
where
\[ ps\text{-}chains\ G \equiv mk\text{-}ps\text{-}chain\ (cNil\ (initial\text{-}ps\ G)) \; lins\ G \]

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

definition \( ps\text{-}chains2 \::\ [\text{node}\ fset,\ \text{diagram}] \Rightarrow ps\text{-}chain\ set \)
where
\[ ps\text{-}chains2\ S\ G \equiv mk\text{-}ps\text{-}chain\ (cNil\ (initial\text{-}ps2\ S\ G)) \; lins2\ S\ G \]

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

lemma \( ps\text{-}chains\text{-}is\text{-}ps\text{-}chains2\text{-}with\text{-}empty\text{-}S \):
\[ ps\text{-}chains = ps\text{-}chains2\ {||} \]
⟨proof⟩

We now wish to describe proofstate chain that are well-formed. First, let us say that \( f \oplus disjoint g \) 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 disjoint [\{|v|\} |\Rightarrow Top\]}, Inl v, \sigma \oplus disjoint [\{|v|\} |\Rightarrow Bot\}])\), where \( v \) is a node, or of the form \((\sigma \oplus disjoint [\{|vs|\} |\Rightarrow Top\}], Inr e, \sigma \oplus disjoint [\{|ws|\} |\Rightarrow Bot\})\), where \( e \) is an edge with source and target nodes \( vs \) and \( ws \) respectively.

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

definition \( wf\text{-}ps\text{-}triple \:: proofstate \times (node + edge) \times proofstate \Rightarrow bool \)
where
\[ wf\text{-}ps\text{-}triple\ T \; (\text{case}\ fnd3\ T\ of) \]
\[ Inl v \Rightarrow (\exists\ \sigma.\ v \notin fdom\ \sigma) \wedge \]
\[ \text{fst3}\ T = [\{|v|\} |\Rightarrow Top\] \oplus \sigma \]
\[ \text{thd3}\ T = [\{|v|\} |\Rightarrow Bot\] \oplus \sigma \]
\[ \text{| Inr} e \Rightarrow (\exists\ \sigma.\ (\text{fst3} e \cup\ \text{thd3} e) \cap fdom\ \sigma = {||}) \wedge \]
\[ \text{fst3}\ T = [\text{fst3} e |\Rightarrow Bot\] \oplus \sigma \]
\[ \text{thd3}\ T = [\text{thd3} e |\Rightarrow Top\] \oplus \sigma) \]

lemma \( wf\text{-}ps\text{-}triple\text{-}nodeI \):
assumes \( \exists\ \sigma.\ v \notin fdom\ \sigma \wedge \)
\[ \sigma 1 = [\{|v|\} |\Rightarrow Top\] \oplus \sigma \wedge \]
\[ \sigma 2 = [\{|v|\} |\Rightarrow Bot\] \oplus \sigma \]
shows \( wf\text{-}ps\text{-}triple\ (\sigma 1,\ \text{Inl} v,\ \sigma 2) \)
⟨proof⟩

lemma \( wf\text{-}ps\text{-}triple\text{-}edgeI \):
assumes \( \exists\ \sigma.\ (\text{fst3} e \cup\ \text{thd3} e) \cap fdom\ \sigma = {||}) \wedge \)
\[ \sigma 1 = [\text{fst3} e |\Rightarrow Bot\] \oplus \sigma \wedge \]
\[ \sigma 2 = [\text{thd3} e |\Rightarrow Top\] \oplus \sigma \]
shows \( wf\text{-}ps\text{-}triple\ (\sigma 1,\ \text{Inr} e,\ \sigma 2) \)

27
definition
wf-ps-chain :: ps-chain ⇒ bool
where
wf-ps-chain ≡ chain-all wf-ps-triple

lemma next-initial-ps2-vertex:
  initial-ps2 ({{v}} ∪ S) G
= initial-ps2 S G ⊕ {{v}} ⊕ [{v}] => Bot
(proof)

lemma next-initial-ps2-edge:
  assumes G = Graph V Λ E and G’ = Graph V’ Λ E’ and
  V’ = V - fst3 e and E’ = removeAll e E and e ∈ set E and
  fst3 e |⊆| S and S |⊆| initials G and wf-dia G
  shows initial-ps2 (S - fst3 e) G’ =
  initial-ps2 S G ⊕ fst3 e ⊕ [thd3 e => Top ]
(proof)

lemma next-lins2-vertex:
  assumes Inl v ≠ π ∈ lins2 S G
  assumes v ∈ S
  shows π ∈ lins2 ({{v}} ∪ S) G
(proof)

lemma next-lins2-edge:
  assumes Inr e ≠ π ∈ lins2 S (Graph V Λ E)
  and vs |⊆| S
  and e = (vs,c,ws)
  shows π ∈ lins2 (S - vs) (Graph (V - vs) Λ (removeAll e E))
(proof)

We wish to prove that every proofstate chain that can be obtained from a
linear extension of G is well-formed and has as its final proofstate that state
in which every terminal node in G is mapped to Bot.

We first prove this for partially-processed diagrams, for then the result for
ordinary diagrams follows as an easy corollary.

We use induction on the size of the partially-processed diagram. The size
of a partially-processed diagram (G, S) is defined as the number of nodes
in G, plus the number of edges, minus the number of nodes in S.

lemmas [simp] = fmember.rep-eq

lemma wf-chains2:
  fixes k
  assumes S |⊆| initials G
  and wf-dia G
  and II ∈ ps-chains2 S G
and \( \text{fcard } G^* + \text{length } G^* E = k + \text{fcard } S \)
shows \( \text{wf-ps-chain } \Pi \land (\text{post } \Pi = [ \text{terminals } G \implies \text{Bot }]) \)
⟨proof⟩

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

9.2 Interface chains

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

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

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

definition \( \text{ps-to-int } :: [\text{diagram}, \text{proofstate}] \Rightarrow \text{interface} \)
where
\( \text{ps-to-int } G \sigma \equiv \bigotimes v \in \text{fdom } \sigma. \text{case-topbot top-ass bot-ass (lookup } \sigma v) (G^* \Lambda v) \)
definition \( \text{ps-chain-to-int-chain } :: [\text{diagram}, \text{ps-chain}] \Rightarrow \text{int-chain} \)
where
\( \text{ps-chain-to-int-chain } G \Pi \equiv \text{chainmap (ps-to-int } G) ((\text{case-sum (Inl } \circ G^* \Lambda) (\text{Inr } \circ \text{snd3}))) \Pi \)
lemma \( \text{ps-chain-to-int-chain-simp} : \text{ps-chain-to-int-chain } (\text{Graph } V \Lambda E) \Pi = \text{chainmap (ps-to-int } (\text{Graph } V \Lambda E)) ((\text{case-sum (Inl } \circ \Lambda) (\text{Inr } \circ \text{snd3}))) \Pi \)  
⟨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 $(\text{asn } P, c, \text{asn } Q)$ 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 $(\text{asn } P, c, \text{asn } Q)$ 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**

$(\text{prov-dia } G \ P \ Q \rightarrow$
\begin{align*}
(P &= \text{top-dia } G \land Q = \text{bot-dia } G \land \\
(\forall c. \text{coms-dia } G \ c \rightarrow \text{prov-triple } (\text{asn } P, c, \text{asn } Q))) \\
\land (\text{prov-com } C \ P \ Q \rightarrow \\
(\forall c. \text{coms-com } C \ c \rightarrow \text{prov-triple } (\text{asn } P, c, \text{asn } Q))) \\
\land (\text{prov-ass } A \rightarrow \\
(\forall c. \text{coms-ass } A \ c \rightarrow \text{prov-triple } (\text{asn } (\text{top-ass } A), c, \text{asn } (\text{bot-ass } A)))))
\end{align*}$

⟨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** $\text{prov-dia } G \ P \ Q$

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

⟨proof⟩

end

References


