Separata: Isabelle tactics for Separation Algebra

By Zhé Hóu, David Sanán, Alwen Tiu, Rajeev Goré, Ranald Clouston

March 19, 2025

Abstract

We bring the labelled sequent calculus LS_{PASL} for propositional abstract separation logic to Isabelle. The tactics given here are directly applied on an extension of the separation algebra in the AFP. In addition to the cancellative separation algebra, we further consider some useful properties in the heap model of separation logic, such as indivisible unit, disjointness, and cross-split. The tactics are essentially a proof search procedure for the calculus LS_{PASL}. We wrap the tactics in an Isabelle method called separata, and give a few examples of separation logic formulae which are provable by separata.

Contents

theory *Separata*
imports *Main Separation-Algebra.Separation-Algebra HOL−Eisbach.Eisbach-Tools HOL−Library.Multiset*
begin

The tactics in this file are a simple proof search procedure based on the labelled sequent calculus LS_PASL for Propositional Abstract Separation Logic in Zhe's PhD thesis.

We define a class which is an extension to cancellative_sep_algebra with other useful properties in separation algebra, including: indivisible unit, disjointness, and cross-split. We also add a property about the (reverse) distributivity of the disjointness.

class *heap-sep-algebra* = *cancellative-sep-algebra* +

assumes *sep-add-ind-unit*: $[\![x + y = 0; x \mathbin{\#\#} y]\!] \Longrightarrow x = 0$
assumes *sep-add-disj*: $x \mathbin{\#\#} x \Longrightarrow x = 0$
assumes *sep-add-cross-split*:
$[\![a + b = w; c + d = w; a \mathbin{\#\#} b; c \mathbin{\#\#} d]\!] \Longrightarrow$
$\exists\ e\ f\ g\ h.\ e + f = a \wedge g + h = b \wedge e + g = c \wedge f + h = d \wedge$
$e \mathbin{\#\#} f \wedge g \mathbin{\#\#} h \wedge e \mathbin{\#\#} g \wedge f \mathbin{\#\#} h$
assumes *disj-dstri*: $[\![x \mathbin{\#\#} y; y \mathbin{\#\#} z; x \mathbin{\#\#} z]\!] \Longrightarrow x \mathbin{\#\#} (y + z)$
begin

1 Lemmas about the labelled sequent calculus.

An abbreviation of the + and ## operators in Separation_Algebra.thy. This notion is closer to the ternary relational atoms used in the literature. This will be the main data structure which our labelled sequent calculus works on.

definition *tern-rel*:: $'a \Rightarrow 'a \Rightarrow 'a \Rightarrow bool$ ($\langle$(-,-$\triangleright$-)$\rangle$ *25*) **where**
tern-rel a b c $\equiv a \mathbin{\#\#} b \wedge a + b = c$

lemma *exist-comb*: $x \mathbin{\#\#} y \Longrightarrow \exists z.\ (x,y \triangleright z)$
by (*simp add*: *tern-rel-def*)

lemma *disj-comb*:
assumes *a1*: $(x,y \triangleright z)$
assumes *a2*: $x \mathbin{\#\#} w$
assumes *a3*: $y \mathbin{\#\#} w$
shows $z \mathbin{\#\#} w$
proof −
from *a1* **have** *f1*: $x \mathbin{\#\#} y \wedge x + y = z$
by (*simp add*: *tern-rel-def*)
then show *?thesis* **using** *a2 a3*
using *local.disj-dstri local.sep-disj-commuteI* **by** *blast*
qed

The following lemmas corresponds to inference rules in LS_PASL. Thus these lemmas prove the soundness of LS_PASL. We also show the invertibility of those rules.

lemma (**in** −) *lspasl-id*:
Gamma $\wedge$ (*A h*) $\Longrightarrow$ (*A h*) $\vee$ *Delta*
by *simp*

lemma (**in** −) *lspasl-botl*:
Gamma $\wedge$ (*sep-false h*) $\Longrightarrow$ *Delta*
by *simp*

lemma (**in** −) *lspasl-topr*:
gamma $\Longrightarrow$ (*sep-true h*) $\vee$ *Delta*
by *simp*

lemma *lspasl-empl*:
Gamma $\wedge$ (*h* = *0*) $\longrightarrow$ *Delta* $\Longrightarrow$
Gamma $\wedge$ (*sep-empty h*) $\longrightarrow$ *Delta*
by (*simp add*: *local.sep-empty-def*)

lemma *lspasl-empl-inv*:
Gamma $\wedge$ (*sep-empty h*) $\longrightarrow$ *Delta* $\Longrightarrow$
Gamma $\wedge$ (*h* = *0*) $\longrightarrow$ *Delta*
by *simp*

The following two lemmas are the same as applying simp add: sep_empty_def.

lemma *lspasl-empl-der*: *sep-empty h* $\Longrightarrow$ *h* = *0*
by (*simp add*: *local.sep-empty-def*)

lemma *lspasl-empl-eq*: (*sep-empty h*) = (*h* = *0*)
by (*simp add*: *local.sep-empty-def*)

lemma *lspasl-empr*:
Gamma $\longrightarrow$ (*sep-empty 0*) $\vee$ *Delta*
by *simp*

end

lemma *lspasl-notl*:
Gamma $\longrightarrow$ (*A h*) $\vee$ *Delta* $\Longrightarrow$
Gamma $\wedge$ ((*not A*) *h*) $\longrightarrow$ *Delta*
by *auto*

lemma *lspasl-notl-inv*:
Gamma $\wedge$ ((*not A*) *h*) $\longrightarrow$ *Delta* $\Longrightarrow$
Gamma $\longrightarrow$ (*A h*) $\vee$ *Delta*
by *auto*

lemma *lspasl-notr*:
Gamma $\wedge$ (*A h*) $\longrightarrow$ *Delta* $\Longrightarrow$
Gamma $\longrightarrow$ ((*not A*) *h*) $\vee$ *Delta*
by *simp*

lemma *lspasl-notr-inv*:
Gamma $\longrightarrow$ ((*not A*) *h*) $\vee$ *Delta* $\Longrightarrow$
Gamma $\wedge$ (*A h*) $\longrightarrow$ *Delta*
by *simp*

lemma *lspasl-andl*:
Gamma $\wedge$ (*A h*) $\wedge$ (*B h*) $\longrightarrow$ *Delta* $\Longrightarrow$
Gamma $\wedge$ ((*A and B*) *h*) $\longrightarrow$ *Delta*
by *simp*

lemma *lspasl-andl-inv*:
$Gamma \wedge ((A\ and\ B)\ h) \longrightarrow Delta \Longrightarrow$
$Gamma \wedge (A\ h) \wedge (B\ h) \longrightarrow Delta$
by *simp*

lemma *lspasl-andr*:
$[\![Gamma \longrightarrow (A\ h) \vee Delta;\ Gamma \longrightarrow (B\ h) \vee Delta]\!] \Longrightarrow$
$Gamma \longrightarrow ((A\ and\ B)\ h) \vee Delta$
by *auto*

lemma *lspasl-andr-inv*:
$Gamma \longrightarrow ((A\ and\ B)\ h) \vee Delta \Longrightarrow$
$(Gamma \longrightarrow (A\ h) \vee Delta) \wedge (Gamma \longrightarrow (B\ h) \vee Delta)$
by *auto*

lemma *lspasl-orl*:
$[\![Gamma \wedge (A\ h) \longrightarrow Delta;\ Gamma \wedge (B\ h) \longrightarrow Delta]\!] \Longrightarrow$
$Gamma \wedge (A\ or\ B)\ h \longrightarrow Delta$
by *auto*

lemma *lspasl-orl-inv*:
$Gamma \wedge (A\ or\ B)\ h \longrightarrow Delta \Longrightarrow$
$(Gamma \wedge (A\ h) \longrightarrow Delta) \wedge (Gamma \wedge (B\ h) \longrightarrow Delta)$
by *simp*

lemma *lspasl-orr*:
$Gamma \longrightarrow (A\ h) \vee (B\ h) \vee Delta \Longrightarrow$
$Gamma \longrightarrow ((A\ or\ B)\ h) \vee Delta$
by *simp*

lemma *lspasl-orr-inv*:
$Gamma \longrightarrow ((A\ or\ B)\ h) \vee Delta \Longrightarrow$
$Gamma \longrightarrow (A\ h) \vee (B\ h) \vee Delta$
by *simp*

lemma *lspasl-impl*:
$[\![Gamma \longrightarrow (A\ h) \vee Delta;\ Gamma \wedge (B\ h) \longrightarrow Delta]\!] \Longrightarrow$
$Gamma \wedge ((A\ imp\ B)\ h) \longrightarrow Delta$
by *auto*

lemma *lspasl-impl-inv*:
$Gamma \wedge ((A\ imp\ B)\ h) \longrightarrow Delta \Longrightarrow$
$(Gamma \longrightarrow (A\ h) \vee Delta) \wedge (Gamma \wedge (B\ h) \longrightarrow Delta)$
by *auto*

lemma *lspasl-impr*:
$Gamma \wedge (A\ h) \longrightarrow (B\ h) \vee Delta \Longrightarrow$
$Gamma \longrightarrow ((A\ imp\ B)\ h) \vee Delta$
by *simp*

lemma *lspasl-impr-inv*:
$Gamma \longrightarrow ((A\ imp\ B)\ h) \vee Delta \Longrightarrow$
$Gamma \wedge (A\ h) \longrightarrow (B\ h) \vee Delta$
by *simp*

context *heap-sep-algebra*
begin

We don't provide lemmas for derivations for the classical connectives, as Isabelle proof methods can easily deal with them.

lemma *lspasl-starl*:
$(\exists\, h1\ h2.\ (Gamma \wedge (h1,h2 \triangleright h0) \wedge (A\ h1) \wedge (B\ h2))) \longrightarrow Delta \Longrightarrow$
$Gamma \wedge ((A \ast\ast B)\ h0) \longrightarrow Delta$
using *local.sep-conj-def* **by** (*auto simp add*: *tern-rel-def*)

lemma *lspasl-starl-inv*:
$Gamma \wedge ((A \ast\ast B)\ h0) \longrightarrow Delta \Longrightarrow$
$(\exists\, h1\ h2.\ (Gamma \wedge (h1,h2 \triangleright h0) \wedge (A\ h1) \wedge (B\ h2))) \longrightarrow Delta$
using *local.sep-conjI* **by** (*auto simp add*: *tern-rel-def*)

lemma *lspasl-starl-der*:
$((A \ast\ast B)\ h0) \Longrightarrow (\exists\, h1\ h2.\ (h1,h2 \triangleright h0) \wedge (A\ h1) \wedge (B\ h2))$
by (*metis lspasl-starl*)

lemma *lspasl-starl-eq*:
$((A \ast\ast B)\ h0) = (\exists\, h1\ h2.\ (h1,h2 \triangleright h0) \wedge (A\ h1) \wedge (B\ h2))$
by (*metis lspasl-starl lspasl-starl-inv*)

lemma *lspasl-starr*:
$[\![Gamma \wedge (h1,h2 \triangleright h0) \longrightarrow (A\ h1) \vee ((A \ast\ast B)\ h0) \vee Delta;$
$Gamma \wedge (h1,h2 \triangleright h0) \longrightarrow (B\ h2) \vee ((A \ast\ast B)\ h0) \vee Delta]\!] \Longrightarrow$
$Gamma \wedge (h1,h2 \triangleright h0) \longrightarrow ((A \ast\ast B)\ h0) \vee Delta$
using *local.sep-conjI* **by** (*auto simp add*: *tern-rel-def*)

lemma *lspasl-starr-inv*:
$Gamma \wedge (h1,h2 \triangleright h0) \longrightarrow ((A \ast\ast B)\ h0) \vee Delta \Longrightarrow$
$(Gamma \wedge (h1,h2 \triangleright h0) \longrightarrow (A\ h1) \vee ((A \ast\ast B)\ h0) \vee Delta) \wedge$
$(Gamma \wedge (h1,h2 \triangleright h0) \longrightarrow (B\ h2) \vee ((A \ast\ast B)\ h0) \vee Delta)$
by *simp*

For efficiency we only apply *R on a pair of a ternary relational atom and a formula ONCE. To achieve this, we create a special predicate to indicate that a pair of a ternary relational atom and a formula has already been used in a *R application. Note that the predicate is true even if the *R rule hasn't been applied. We will not infer the truth of this predicate in proof search, but only check its syntactical appearance, which is only generated by the lemma lspasl_starr_der. We need to ensure that this predicate is not generated elsewhere in the proof search.

definition *starr-applied*:: $'a \Rightarrow 'a \Rightarrow 'a \Rightarrow ('a \Rightarrow bool) \Rightarrow bool$ **where**
starr-applied h1 h2 h0 F $\equiv$ (*h1*,*h2*$\triangleright$*h0*) $\wedge$ $\neg$(*F h0*)

lemma *lspasl-starr-der*:
(*h1*,*h2*$\triangleright$*h0*) $\Longrightarrow$ $\neg$ ((*A* $**$ *B*) *h0*) $\Longrightarrow$
((*h1*,*h2*$\triangleright$*h0*) $\wedge$ $\neg$ ((*A h1*) $\vee$ ((*A* $**$ *B*) *h0*)) $\wedge$ (*starr-applied h1 h2 h0* (*A* $**$ *B*)))
$\vee$
((*h1*,*h2*$\triangleright$*h0*) $\wedge$ $\neg$ ((*B h2*) $\vee$ ((*A* $**$ *B*) *h0*)) $\wedge$ (*starr-applied h1 h2 h0* (*A* $**$ *B*)))
by (*simp add*: *lspasl-starl-eq starr-applied-def*)

lemma *lspasl-starr-eq*:
((*h1*,*h2*$\triangleright$*h0*) $\wedge$ $\neg$ ((*A* $**$ *B*) *h0*)) =
(((*h1*,*h2*$\triangleright$*h0*) $\wedge$ $\neg$ ((*A h1*) $\vee$ ((*A* $**$ *B*) *h0*))) $\vee$ ((*h1*,*h2*$\triangleright$*h0*) $\wedge$ $\neg$ ((*B h2*) $\vee$ ((*A* $**$ *B*) *h0*))))
using *lspasl-starr-der* **by** *blast*

lemma *lspasl-magicl*:
⟦*Gamma* $\wedge$ (*h1*,*h2*$\triangleright$*h0*) $\wedge$ ((*A* $\longrightarrow*$ *B*) *h2*) $\longrightarrow$ (*A h1*) $\vee$ *Delta*;
Gamma $\wedge$ (*h1*,*h2*$\triangleright$*h0*) $\wedge$ ((*A* $\longrightarrow*$ *B*) *h2*) $\wedge$ (*B h0*) $\longrightarrow$ *Delta*⟧ $\Longrightarrow$
Gamma $\wedge$ (*h1*,*h2*$\triangleright$*h0*) $\wedge$ ((*A* $\longrightarrow*$ *B*) *h2*) $\longrightarrow$ *Delta*
using *local.sep-add-commute local.sep-disj-commuteI local.sep-implD tern-rel-def*
by *fastforce*

lemma *lspasl-magicl-inv*:
Gamma $\wedge$ (*h1*,*h2*$\triangleright$*h0*) $\wedge$ ((*A* $\longrightarrow*$ *B*) *h2*) $\longrightarrow$ *Delta* $\Longrightarrow$
(*Gamma* $\wedge$ (*h1*,*h2*$\triangleright$*h0*) $\wedge$ ((*A* $\longrightarrow*$ *B*) *h2*) $\longrightarrow$ (*A h1*) $\vee$ *Delta*) $\wedge$
(*Gamma* $\wedge$ (*h1*,*h2*$\triangleright$*h0*) $\wedge$ ((*A* $\longrightarrow*$ *B*) *h2*) $\wedge$ (*B h0*) $\longrightarrow$ *Delta*)
by *simp*

For efficiency we only apply -*L on a pair of a ternary relational atom and a formula ONCE. To achieve this, we create a special predicate to indicate that a pair of a ternary relational atom and a formula has already been used in a *R application. Note that the predicate is true even if the *R rule hasn't been applied. We will not infer the truth of this predicate in proof search, but only check its syntactical appearance, which is only generated by the lemma lspasl_magicl_der. We need to ensure that in the proof search of Separata, this predicate is not generated elsewhere.

definition *magicl-applied*:: $'a \Rightarrow 'a \Rightarrow 'a \Rightarrow ('a \Rightarrow bool) \Rightarrow bool$ **where**
magicl-applied h1 h2 h0 F $\equiv$ (*h1*,*h2*$\triangleright$*h0*) $\wedge$ (*F h2*)

lemma *lspasl-magicl-der*:
(*h1*,*h2*$\triangleright$*h0*) $\Longrightarrow$ ((*A* $\longrightarrow*$ *B*) *h2*) $\Longrightarrow$
((*h1*,*h2*$\triangleright$*h0*) $\wedge$ $\neg$(*A h1*) $\wedge$ ((*A* $\longrightarrow*$ *B*) *h2*) $\wedge$ (*magicl-applied h1 h2 h0* (*A* $\longrightarrow*$ *B*))) $\vee$
((*h1*,*h2*$\triangleright$*h0*) $\wedge$ (*B h0*) $\wedge$ ((*A* $\longrightarrow*$ *B*) *h2*) $\wedge$ (*magicl-applied h1 h2 h0* (*A* $\longrightarrow*$ *B*)))
by (*metis lspasl-magicl magicl-applied-def*)

lemma *lspasl-magicl-eq*:
(*h1*,*h2*▷*h0*) ∧ ((*A* ⟶∗ *B*) *h2*)) =
(((*h1*,*h2*▷*h0*) ∧ ¬(*A h1*) ∧ ((*A* ⟶∗ *B*) *h2*)) ∨ ((*h1*,*h2*▷*h0*) ∧ (*B h0*) ∧ ((*A* ⟶∗ *B*) *h2*)))
using *lspasl-magicl-der* **by** *blast*

lemma *lspasl-magicr*:
(∃ *h1 h0*. *Gamma* ∧ (*h1*,*h2*▷*h0*) ∧ (*A h1*) ∧ ((*not B*) *h0*)) ⟶ *Delta* ⟹
Gamma ⟶ ((*A* ⟶∗ *B*) *h2*) ∨ *Delta*
using *local.sep-add-commute local.sep-disj-commute local.sep-impl-def tern-rel-def*
by *auto*

lemma *lspasl-magicr-inv*:
Gamma ⟶ ((*A* ⟶∗ *B*) *h2*) ∨ *Delta* ⟹
(∃ *h1 h0*. *Gamma* ∧ (*h1*,*h2*▷*h0*) ∧ (*A h1*) ∧ ((*not B*) *h0*)) ⟶ *Delta*
by (*metis lspasl-magicl*)

lemma *lspasl-magicr-der*:
¬ ((*A* ⟶∗ *B*) *h2*) ⟹
(∃ *h1 h0*. (*h1*,*h2*▷*h0*) ∧ (*A h1*) ∧ ((*not B*) *h0*))
by (*metis lspasl-magicr*)

lemma *lspasl-magicr-eq*:
(¬ ((*A* ⟶∗ *B*) *h2*)) =
((∃ *h1 h0*. (*h1*,*h2*▷*h0*) ∧ (*A h1*) ∧ ((*not B*) *h0*)))
by (*metis lspasl-magicl lspasl-magicr*)

lemma *lspasl-eq*:
Gamma ∧ (*0*,*h2*▷*h2*) ∧ *h1* = *h2* ⟶ *Delta* ⟹
Gamma ∧ (*0*,*h1*▷*h2*) ⟶ *Delta*
by (*simp add*: *tern-rel-def*)

lemma *lspasl-eq-inv*:
Gamma ∧ (*0*,*h1*▷*h2*) ⟶ *Delta* ⟹
Gamma ∧ (*0*,*h2*▷*h2*) ∧ *h1* = *h2* ⟶ *Delta*
by *simp*

lemma *lspasl-eq-der*: (*0*,*h1*▷*h2*) ⟹ ((*0*,*h1*▷*h1*) ∧ *h1* = *h2*)
using *lspasl-eq* **by** *auto*

lemma *lspasl-eq-eq*: (*0*,*h1*▷*h2*) = ((*0*,*h1*▷*h1*) ∧ (*h1* = *h2*))
by (*simp add*: *tern-rel-def*)

lemma *lspasl-u*:
Gamma ∧ (*h*,*0*▷*h*) ⟶ *Delta* ⟹
Gamma ⟶ *Delta*
by (*simp add*: *tern-rel-def*)

lemma *lspasl-u-inv*:
Gamma ⟶ *Delta* ⟹
Gamma ∧ (*h*,*0*▷*h*) ⟶ *Delta*
by *simp*

lemma *lspasl-u-der*: (*h*,*0*▷*h*)
using *lspasl-u* **by** *auto*

lemma *lspasl-e*:
Gamma ∧ (*h1*,*h2*▷*h0*) ∧ (*h2*,*h1*▷*h0*) ⟶ *Delta* ⟹
Gamma ∧ (*h1*,*h2*▷*h0*) ⟶ *Delta*
by (*simp add*: *local.sep-add-commute local.sep-disj-commute tern-rel-def*)

lemma *lspasl-e-inv*:
Gamma ∧ (*h1*,*h2*▷*h0*) ⟶ *Delta* ⟹
Gamma ∧ (*h1*,*h2*▷*h0*) ∧ (*h2*,*h1*▷*h0*) ⟶ *Delta*
by *simp*

lemma *lspasl-e-der*: (*h1*,*h2*▷*h0*) ⟹ (*h1*,*h2*▷*h0*) ∧ (*h2*,*h1*▷*h0*)
using *lspasl-e* **by** *blast*

lemma *lspasl-e-eq*: (*h1*,*h2*▷*h0*) = ((*h1*,*h2*▷*h0*) ∧ (*h2*,*h1*▷*h0*))
using *lspasl-e* **by** *blast*

lemma *lspasl-a-der*:
assumes *a1*: (*h1*,*h2*▷*h0*)
and *a2*: (*h3*,*h4*▷*h1*)
shows (∃ *h5*. (*h3*,*h5*▷*h0*) ∧ (*h2*,*h4*▷*h5*) ∧ (*h1*,*h2*▷*h0*) ∧ (*h3*,*h4*▷*h1*))
proof −
have *f1*: *h1* ## *h2*
using *a1* **by** (*simp add*: *tern-rel-def*)
have *f2*: *h3* ## *h4*
using *a2* **by** (*simp add*: *tern-rel-def*)
have *f3*: *h3* + *h4* = *h1*
using *a2* **by** (*simp add*: *tern-rel-def*)
then have *h3* ## *h2*
using *f2 f1* **by** (*metis local.sep-disj-addD1 local.sep-disj-commute*)
then have *f4*: *h2* ## *h3*
by (*metis local.sep-disj-commute*)
then have *f5*: *h2* + *h4* ## *h3*
using *f3 f2 f1* **by** (*metis* (*no-types*) *local.sep-add-commute local.sep-add-disjI1*)
have *h4* ## *h2*
using *f3 f2 f1* **by** (*metis local.sep-add-commute local.sep-disj-addD1 local.sep-disj-commute*)
then show *?thesis*
using *f5 f4* **by** (*metis* (*no-types*) *assms tern-rel-def local.sep-add-assoc local.sep-add-commute local.sep-disj-commute*)
qed

lemma *lspasl-a*:

$(\exists\, h5.\ Gamma \wedge (h3,h5 \triangleright h0) \wedge (h2,h4 \triangleright h5) \wedge (h1,h2 \triangleright h0) \wedge (h3,h4 \triangleright h1)) \longrightarrow Delta \Longrightarrow$
$Gamma \wedge (h1,h2 \triangleright h0) \wedge (h3,h4 \triangleright h1) \longrightarrow Delta$
using *lspasl-a-der* **by** *blast*

lemma *lspasl-a-inv*:
$Gamma \wedge (h1,h2 \triangleright h0) \wedge (h3,h4 \triangleright h1) \longrightarrow Delta \Longrightarrow$
$(\exists\, h5.\ Gamma \wedge (h3,h5 \triangleright h0) \wedge (h2,h4 \triangleright h5) \wedge (h1,h2 \triangleright h0) \wedge (h3,h4 \triangleright h1)) \longrightarrow Delta$
by *auto*

lemma *lspasl-a-eq*:
$((h1,h2 \triangleright h0) \wedge (h3,h4 \triangleright h1)) =$
$(\exists\, h5.\ (h3,h5 \triangleright h0) \wedge (h2,h4 \triangleright h5) \wedge (h1,h2 \triangleright h0) \wedge (h3,h4 \triangleright h1))$
using *lspasl-a-der* **by** *blast*

lemma *lspasl-p*:
$Gamma \wedge (h1,h2 \triangleright h0) \wedge h0 = h3 \longrightarrow Delta \Longrightarrow$
$Gamma \wedge (h1,h2 \triangleright h0) \wedge (h1,h2 \triangleright h3) \longrightarrow Delta$
by (*auto simp add*: *tern-rel-def*)

lemma *lspasl-p-inv*:
$Gamma \wedge (h1,h2 \triangleright h0) \wedge (h1,h2 \triangleright h3) \longrightarrow Delta \Longrightarrow$
$Gamma \wedge (h1,h2 \triangleright h0) \wedge h0 = h3 \longrightarrow Delta$
by *auto*

lemma *lspasl-p-der*:
$(h1,h2 \triangleright h0) \Longrightarrow (h1,h2 \triangleright h3) \Longrightarrow (h1,h2 \triangleright h0) \wedge h0 = h3$
by (*simp add*: *tern-rel-def*)

lemma *lspasl-p-eq*:
$((h1,h2 \triangleright h0) \wedge (h1,h2 \triangleright h3)) = ((h1,h2 \triangleright h0) \wedge h0 = h3)$
using *lspasl-p-der* **by** *auto*

lemma *lspasl-c*:
$Gamma \wedge (h1,h2 \triangleright h0) \wedge h2 = h3 \longrightarrow Delta \Longrightarrow$
$Gamma \wedge (h1,h2 \triangleright h0) \wedge (h1,h3 \triangleright h0) \longrightarrow Delta$
by (*metis local.sep-add-cancelD local.sep-add-commute tern-rel-def local.sep-disj-commuteI*)

lemma *lspasl-c-inv*:
$Gamma \wedge (h1,h2 \triangleright h0) \wedge (h1,h3 \triangleright h0) \longrightarrow Delta \Longrightarrow$
$Gamma \wedge (h1,h2 \triangleright h0) \wedge h2 = h3 \longrightarrow Delta$
by *auto*

lemma *lspasl-c-der*:
$(h1,h2 \triangleright h0) \Longrightarrow (h1,h3 \triangleright h0) \Longrightarrow (h1,h2 \triangleright h0) \wedge h2 = h3$
using *lspasl-c* **by** *blast*

lemma *lspasl-c-eq*:
(*h1*,*h2*▷*h0*) ∧ (*h1*,*h3*▷*h0*)) = ((*h1*,*h2*▷*h0*) ∧ *h2* = *h3*)
using *lspasl-c-der* **by** *auto*

lemma *lspasl-iu*:
Gamma ∧ (*0*,*h2*▷*0*) ∧ *h1* = *0* ⟶ *Delta* ⟹
Gamma ∧ (*h1*,*h2*▷*0*) ⟶ *Delta*
using *local.sep-add-ind-unit tern-rel-def* **by** *blast*

lemma *lspasl-iu-inv*:
Gamma ∧ (*h1*,*h2*▷*0*) ⟶ *Delta* ⟹
Gamma ∧ (*0*,*h2*▷*0*) ∧ *h1* = *0* ⟶ *Delta*
by *simp*

lemma *lspasl-iu-der*:
(*h1*,*h2*▷*0*) ⟹ ((*0*,*0*▷*0*) ∧ *h1* = *0* ∧ *h2* = *0*)
using *lspasl-eq-der lspasl-iu* **by** (*auto simp add*: *tern-rel-def*)

lemma *lspasl-iu-eq*:
(*h1*,*h2*▷*0*) = ((*0*,*0*▷*0*) ∧ *h1* = *0* ∧ *h2* = *0*)
using *lspasl-iu-der* **by** *blast*

lemma *lspasl-d*:
Gamma ∧ (*0*,*0*▷*h2*) ∧ *h1* = *0* ⟶ *Delta* ⟹
Gamma ∧ (*h1*,*h1*▷*h2*) ⟶ *Delta*
using *local.sep-add-disj tern-rel-def* **by** *fastforce*

lemma *lspasl-d-inv*:
Gamma ∧ (*h1*,*h1*▷*h2*) ⟶ *Delta* ⟹
Gamma ∧ (*0*,*0*▷*h2*) ∧ *h1* = *0* ⟶ *Delta*
by *blast*

lemma *lspasl-d-der*:
(*h1*,*h1*▷*h2*) ⟹ (*0*,*0*▷*0*) ∧ *h1* = *0* ∧ *h2* = *0*
using *lspasl-d lspasl-eq-der* **by** *blast*

lemma *lspasl-d-eq*:
(*h1*,*h1*▷*h2*) = ((*0*,*0*▷*0*) ∧ *h1* = *0* ∧ *h2* = *0*)
using *lspasl-d-der* **by** *blast*

lemma *lspasl-cs-der*:
assumes *a1*: (*h1*,*h2*▷*h0*)
and *a2*: (*h3*,*h4*▷*h0*)
shows (∃ *h5 h6 h7 h8*. (*h5*,*h6*▷*h1*) ∧ (*h7*,*h8*▷*h2*) ∧(*h5*,*h7*▷*h3*) ∧ (*h6*,*h8*▷*h4*)
∧ (*h1*,*h2*▷*h0*) ∧ (*h3*,*h4*▷*h0*))
proof −
from *a1 a2* **have** *h1* + *h2* = *h0* ∧ *h3* + *h4* = *h0* ∧ *h1* ## *h2* ∧ *h3* ## *h4*
by (*simp add*: *tern-rel-def*)
then have ∃ *h5 h6 h7 h8*. *h5* + *h6* = *h1* ∧ *h7* + *h8* = *h2* ∧

h5 + h7 = h3 ∧ h6 + h8 = h4 ∧ h5 ## h6 ∧ h7 ## h8 ∧
h5 ## h7 ∧ h6 ## h8
using *local.sep-add-cross-split* **by** *auto*
then have ∃ *h5 h6 h7 h8*. *(h5,h6▷h1) ∧ h7 + h8 = h2 ∧*
h5 + h7 = h3 ∧ h6 + h8 = h4 ∧ h7 ## h8 ∧
h5 ## h7 ∧ h6 ## h8
by (*auto simp add*: *tern-rel-def*)
then have ∃ *h5 h6 h7 h8*. *(h5,h6▷h1) ∧ (h7,h8▷h2) ∧*
h5 + h7 = h3 ∧ h6 + h8 = h4 ∧ h5 ## h7 ∧ h6 ## h8
by (*auto simp add*: *tern-rel-def*)
then have ∃ *h5 h6 h7 h8*. *(h5,h6▷h1) ∧ (h7,h8▷h2) ∧*
(h5,h7▷h3) ∧ h6 + h8 = h4 ∧ h6 ## h8
by (*auto simp add*: *tern-rel-def*)
then show *?thesis* **using** *a1 a2 tern-rel-def* **by** *blast*
qed

lemma *lspasl-cs*:
(∃ *h5 h6 h7 h8*. *Gamma ∧ (h5,h6▷h1) ∧ (h7,h8▷h2) ∧(h5,h7▷h3) ∧ (h6,h8▷h4)*
∧ (h1,h2▷h0) ∧ (h3,h4▷h0)) ⟶ Delta ⟹
Gamma ∧ (h1,h2▷h0) ∧ (h3,h4▷h0) ⟶ Delta
using *lspasl-cs-der* **by** *auto*

lemma *lspasl-cs-inv*:
Gamma ∧ (h1,h2▷h0) ∧ (h3,h4▷h0) ⟶ Delta ⟹
(∃ *h5 h6 h7 h8*. *Gamma ∧ (h5,h6▷h1) ∧ (h7,h8▷h2) ∧(h5,h7▷h3) ∧ (h6,h8▷h4)*
∧ (h1,h2▷h0) ∧ (h3,h4▷h0)) ⟶ Delta
by *auto*

lemma *lspasl-cs-eq*:
((h1,h2▷h0) ∧ (h3,h4▷h0)) =
(∃ h5 h6 h7 h8. *(h5,h6▷h1) ∧ (h7,h8▷h2) ∧(h5,h7▷h3) ∧ (h6,h8▷h4) ∧*
(h1,h2▷h0) ∧ (h3,h4▷h0))
using *lspasl-cs-der* **by** *auto*

end

The above proves the soundness and invertibility of LS_PASL.

2 Lemmas David proved for separation algebra.

lemma *sep-substate-tran*:
x ⪯ y ∧ y ⪯ z ⟹ x ⪯ z
unfolding *sep-substate-def*
proof −
assume (∃ *z*. *x ## z ∧ x + z = y) ∧ (∃ za*. *y ## za ∧ y + za = z)*
then obtain *x′ y′* **where** *fixed*:*(x ## x′ ∧ x + x′ = y) ∧ (y ## y′ ∧ y + y′*
= z)
by *auto*
then have *disj-x*:*x ## y′ ∧ x′ ## y′*

using *sep-disj-addD sep-disj-commute* **by** *blast*
then have *p1*:x ## (x' + y') **using** *fixed sep-disj-commute sep-disj-addI3*
by *blast*
then have $x + (x' + y') = z$ **using** *disj-x* **by** (*metis* (*no-types*) *fixed sep-add-assoc*)

thus $\exists$ *za*. x ## *za* $\wedge$ x + *za* = z **using** *p1* **by** *auto*
qed

lemma *precise-sep-conj*:
assumes *a1*:*precise* I **and**
a2:*precise* I'
shows *precise* (I $\wedge*$ I')
proof (*clarsimp simp*: *precise-def*)
fix *hp hp' h*
assume *hp*:*hp* $\preceq$ *h* **and** *hp'*: *hp'* $\preceq$ *h* **and** *ihp*: (I $\wedge*$ I') *hp* **and** *ihp'*: (I $\wedge*$ I') *hp'*
obtain *hp1 hp2* **where** *ihpex*: *hp1* ## *hp2* $\wedge$ *hp* = *hp1* + *hp2* $\wedge$ I *hp1* $\wedge$ I' *hp2* **using** *ihp sep-conjD* **by** *blast*
obtain *hp1' hp2'* **where** *ihpex'*: *hp1'* ## *hp2'* $\wedge$ *hp'* = *hp1'* + *hp2'* $\wedge$ I *hp1'* $\wedge$ I' *hp2'* **using** *ihp' sep-conjD* **by** *blast*
have *f3*: *hp2'* ## *hp1'*
by (*simp add*: *ihpex' sep-disj-commute*)
have *f4*: *hp2* ## *hp1*
using *ihpex sep-disj-commute* **by** *blast*
have *f5*:$\bigwedge a$. $\neg$ a $\preceq$ *hp* $\vee$ a $\preceq$ *h*
using *hp sep-substate-tran* **by** *blast*
have *f6*:$\bigwedge a$. $\neg$ a $\preceq$ *hp'* $\vee$ a $\preceq$ *h*
using *hp' sep-substate-tran* **by** *blast*
thus *hp* = *hp'*
using *f4 f3 f5 a2 a1 a1 a2 ihpex ihpex'*
unfolding *precise-def* **by** (*metis sep-add-commute sep-substate-disj-add'*)
qed

lemma *unique-subheap*:
(σ*1*,σ*2*$\triangleright\sigma$) $\Longrightarrow$ $\exists!\sigma$*2'*.(σ*1*,σ*2'*$\triangleright\sigma$)
using *lspasl-c-der* **by** *blast*

lemma *sep-split-substate*:
(σ*1*, σ*2*$\triangleright$ σ) $\Longrightarrow$
(σ*1* $\preceq$ σ) $\wedge$ (σ*2* $\preceq$ σ)
proof−
assume *a1*:(σ*1*, σ*2*$\triangleright$ σ)
thus (σ*1* $\preceq$ σ) $\wedge$ (σ*2* $\preceq$ σ)
by (*auto simp add*: *sep-disj-commute*
tern-rel-def
sep-substate-disj-add
sep-substate-disj-add')
qed

abbreviation *sep-septraction* :: (('*a*::*sep-algebra*) ⇒ *bool*) ⇒ (*'a* ⇒ *bool*) ⇒ (*'a* ⇒ *bool*) (**infixr** ‹⟶⊕› *25*)
where
P ⟶⊕ *Q* ≡ *not* (*P* ⟶∗ *not Q*)

3 Below we integrate the inference rules in proof search.

method *try-lspasl-empl* = (
match **premises in** *P*[*thin*]:*sep-empty ?h* ⇒
‹*insert lspasl-empl-der*[*OF P*]›,
simp?
)

method *try-lspasl-starl* = (
match **premises in** *P*[*thin*]:(*?A* ∗∗ *?B*) *?h* ⇒
‹*insert lspasl-starl-der*[*OF P*], *auto*›,
simp?
)

method *try-lspasl-magicr* = (
match **premises in** *P*[*thin*]:¬(*?A* ⟶∗ *?B*) *?h* ⇒
‹*insert lspasl-magicr-der*[*OF P*], *auto*›,
simp?
)

Only apply the rule Eq on (0,h1,h2) where h1 and h2 are not syntactically the same.

method *try-lspasl-eq* = (
match **premises in** *P*[*thin*]:(*0*,*?h1*▷*?h2*) ⇒
‹*match P in*
(*0*,*h*▷*h*) *for h* ⇒ ‹*fail*›
|- ⇒ ‹*insert lspasl-eq-der*[*OF P*], *auto*››,
simp?
)

We restrict that the rule IU can't be applied on (0,0,0).

method *try-lspasl-iu* = (
match **premises in** *P*[*thin*]:(*?h1*,*?h2*▷*0*) ⇒
‹*match P in*
(*0*,*0*▷*0*) ⇒ ‹*fail*›
|- ⇒ ‹*insert lspasl-iu-der*[*OF P*], *auto*››,
simp?
)

We restrict that the rule D can't be applied on (0,0,0).

method *try-lspasl-d* = (

match **premises in** *P*[*thin*]:(*h1*,*h1*▷*h2*) **for** *h1 h2* ⇒
‹*match P in*
(*0*,*0*▷*0*) ⇒ ‹*fail*›
|- ⇒ ‹*insert lspasl-d-der*[*OF P*], *auto*››,
simp?
)

We restrict that the rule P can't be applied to two syntactically identical ternary relational atoms.

method *try-lspasl-p* = (
match **premises in** *P*[*thin*]:(*h1*,*h2*▷*h0*) **for** *h0 h1 h2* ⇒
‹*match premises in* (*h1*,*h2*▷*h0*) ⇒ ‹*fail*›
|*P'*[*thin*]:(*h1*,*h2*▷*?h3*) ⇒ ‹*insert lspasl-p-der*[*OF P P'*], *auto*››,
simp?
)

We restrict that the rule C can't be applied to two syntactically identical ternary relational atoms.

method *try-lspasl-c* = (
match **premises in** *P*[*thin*]:(*h1*,*h2*▷*h0*) **for** *h0 h1 h2* ⇒
‹*match premises in* (*h1*,*h2*▷*h0*) ⇒ ‹*fail*›
|*P'*[*thin*]:(*h1*,*?h3*▷*h0*) ⇒ ‹*insert lspasl-c-der*[*OF P P'*], *auto*››,
simp?
)

We restrict that *R only applies to a pair of a ternary relational and a formula once. Here, we need to first try simp to unify heaps. In the end, we try simp_all to simplify all branches. A similar strategy is used in -*L.

method *try-lspasl-starr* = (
simp?,
match **premises in** *P*:(*h1*,*h2*▷*h*) **and** *P'*:¬(*A* ∗∗ *B*) (*h*::*'a*::*heap-sep-algebra*)
for *h1 h2 h A B* ⇒
‹*match premises in starr-applied h1 h2 h* (*A* ∗∗ *B*) ⇒ ‹*fail*›
|- ⇒ ‹*insert lspasl-starr-der*[*OF P P'*], *auto*››,
simp-all?
)

We restrict that -*L only applies to a pair of a ternary relational and a formula once.

method *try-lspasl-magicl* = (
simp?,
match **premises in** *P*: (*h1*,*h*▷*h2*) **and** *P'*:(*A* ⟶∗ *B*) (*h*::*'a*::*heap-sep-algebra*)
for *h1 h2 h A B* ⇒
‹*match premises in magicl-applied h1 h h2* (*A* ⟶∗ *B*) ⇒ ‹*fail*›
|- ⇒ ‹*insert lspasl-magicl-der*[*OF P P'*], *auto*››,
simp-all?
)

We restrict that the U rule is only applicable to a world h when (h,0,h) is not in the premises. There are two cases: (1) We pick a ternary relational atom (h1,h2,h0), and check if (h1,0,h1) occurs in the premises, if not, apply U on h1. Otherwise, check other ternary relational atoms. (2) We pick a labelled formula (A h), and check if (h,0,h) occurs in the premises, if not, apply U on h. Otherwise, check other labelled formulae.

method *try-lspasl-u-tern* = (
match **premises in**
P:(*h1*,*h2*▷(*h0*::'*a*::*heap-sep-algebra*)) **for** *h1 h2 h0* ⇒
‹*match premises in*
(*h1*,*0*▷*h1*) ⇒ ‹*match premises in*
(*h2*,*0*▷*h2*) ⇒ ‹*match premises in*
I1:(*h0*,*0*▷*h0*) ⇒ ‹*fail*›
|- ⇒ ‹*insert lspasl-u-der*[*of h0*]››
|- ⇒ ‹*insert lspasl-u-der*[*of h2*]››
|- ⇒ ‹*insert lspasl-u-der*[*of h1*]››,
simp?
)

method *try-lspasl-u-form* = (
match **premises in**
P':- (*h*::'*a*::*heap-sep-algebra*) **for** *h* ⇒
‹*match premises in* (*h*,*0*▷*h*) ⇒ ‹*fail*›
|(*0*,*0*▷*0*) *and h* = *0* ⇒ ‹*fail*›
|(*0*,*0*▷*0*) *and 0* = *h* ⇒ ‹*fail*›
|- ⇒ ‹*insert lspasl-u-der*[*of h*]››,
simp?
)

We restrict that the E rule is only applicable to (h1,h2,h0) when (h2,h1,h0) is not in the premises.

method *try-lspasl-e* = (
match **premises in** *P*:(*h1*,*h2*▷*h0*) **for** *h1 h2 h0* ⇒
‹*match premises in* (*h2*,*h1*▷*h0*) ⇒ ‹*fail*›
|- ⇒ ‹*insert lspasl-e-der*[*OF P*], *auto*››,
simp?
)

We restrict that the A rule is only applicable to (h1,h2,h0) and (h3,h4,h1) when (h3,h,h0) and (h2,h4,h) or any commutative variants of the two do not occur in the premises, for some h. Additionally, we do not allow A to be applied to two identical ternary relational atoms. We further restrict that the leaves must not be 0, because otherwise this application does not gain anything.

method *try-lspasl-a* = (
match **premises in** (*h1*,*h2*▷*h0*) **for** *h0 h1 h2* ⇒
‹*match premises in*

```
(0,h2▷h0) ⇒ ‹fail›
|(h1,0▷h0) ⇒ ‹fail›
|(h1,h2▷0) ⇒ ‹fail›
|P[thin]:(h1,h2▷h0) ⇒
‹match premises in
P':(h3,h4▷h1) for h3 h4 ⇒ ‹match premises in
(0,h4▷h1) ⇒ ‹fail›
|(h3,0▷h1) ⇒ ‹fail›
|(-,h3▷h0) ⇒ ‹fail›
|(h3,-▷h0) ⇒ ‹fail›
|(h2,h4▷-) ⇒ ‹fail›
|(h4,h2▷-) ⇒ ‹fail›
|- ⇒ ‹insert P P', drule lspasl-a-der, auto››››,
simp?
)
```

I don't have a good heuristics for CS right now. I simply forbid CS to be applied on the same pair twice.

```
method try-lspasl-cs = (
    match premises in P[thin]:(h1,h2▷h0) for h0 h1 h2 ⇒
    ‹match premises in (h1,h2▷h0) ⇒ ‹fail›
    |(h2,h1▷h0) ⇒ ‹fail›
    |P':(h3,h4▷h0) for h3 h4 ⇒ ‹match premises in
    (h5,h6▷h1) and (h7,h8▷h2) and (h5,h7▷h3) and (h6,h8▷h4) for h5 h6 h7 h8
⇒ ‹fail›
    |(i5,i6▷h2) and (i7,i8▷h1) and (i5,i7▷h3) and (i6,i8▷h4) for i5 i6 i7 i8 ⇒
‹fail›
    |(j5,j6▷h1) and (j7,j8▷h2) and (j5,j7▷h4) and (j6,j8▷h3) for j5 j6 j7 j8 ⇒
‹fail›
    |(k5,k6▷h2) and (k7,k8▷h1) and (k5,k7▷h4) and (k6,k8▷h3) for k5 k6 k7 k8
⇒ ‹fail›
    |- ⇒ ‹insert lspasl-cs-der[OF P P'], auto›››,
    simp
    )

method try-lspasl-starr-guided = (
    simp?,
     match premises in P:(h1,h2▷h) and P':¬(A ** B) (h::'a::heap-sep-algebra)
for h1 h2 h A B ⇒
    ‹match premises in starr-applied h1 h2 h (A ** B) ⇒ ‹fail›
    |A h1 ⇒ ‹insert lspasl-starr-der[OF P P'], auto›
    |B h2 ⇒ ‹insert lspasl-starr-der[OF P P'], auto››,
    simp-all?
    )

method try-lspasl-magicl-guided = (
    simp?,
    match premises in P: (h1,h▷h2) and P':(A —→* B) (h::'a::heap-sep-algebra)
for h1 h2 h A B ⇒
```

```
‹match premises in magicl-applied h1 h h2 (A ⟶* B) ⇒ ‹fail›
|A h1 ⇒ ‹insert lspasl-magicl-der[OF P P'], auto›
|¬(B h2) ⇒ ‹insert lspasl-magicl-der[OF P P'], auto››,
simp-all?
)
```

In case the conclusion is not False, we normalise the goal as below.

```
method norm-goal = (
    match conclusion in False ⇒ ‹fail›
    |- ⇒ ‹rule ccontr›,
    simp?
    )
```

The tactic for separata. We first try to simplify the problem with auto simp add: sep_conj_ac, which ought to solve many problems. Then we apply the "true" invertible rules and structural rules which unify worlds as much as possible, followed by auto to simplify the goals. Then we apply *R and -*L and other structural rules. The rule CS is only applied when nothing else is applicable. We try not to use it.

***** Note, (try_lspasl_u |try_lspasl_e) |try_lspasl_a)+ may cause infinite loops. *****

```
method separata =
  ((auto simp add: sep-conj-ac)
    |(norm-goal?,
        ((try-lspasl-empl
          |try-lspasl-starl
          |try-lspasl-magicr
          |try-lspasl-iu
          |try-lspasl-d
          |try-lspasl-eq
          |try-lspasl-p
          |try-lspasl-c
          |try-lspasl-starr-guided
          |try-lspasl-magicl-guided)+,
          auto?)
      |(try-lspasl-u-tern
        |try-lspasl-e
        |try-lspasl-a)+
      |(try-lspasl-starr
        |try-lspasl-magicl)
        )+
    |try-lspasl-u-form+
    |try-lspasl-cs
      )+
```

4 Some examples.

Let's prove something that abstract separation logic provers struggle to prove. This can be proved easily in Isabelle, proof found by Sledgehammer.

lemma *fm-hard*: ((*sep-empty imp* (*p0* $\longrightarrow*$ (((*p0* $**$ (*p0* $\longrightarrow*$ *p1*)) $**$ (*not p1*)) $\longrightarrow*$
(*p0* $**$ (*p0* $**$ ((*p0* $\longrightarrow*$ *p1*) $**$ (*not p1*)))))))) *imp* ((((*sep-empty* $**$ *p0*) $**$
(*p0* $**$ ((*p0* $\longrightarrow*$ *p1*) $**$ (*not p1*)))) *imp* (((*p0* $**$ *p0*) $**$ (*p0* $\longrightarrow*$ *p1*)) $**$
(*not p1*))) $**$ *sep-empty*)) *h*
by *separata*

The following formula can only be proved in partial-deterministic separation algebras. Sledgehammer took a rather long time to find a proof.

lemma *fm-partial*: (((*not* (*sep-true* $\longrightarrow*$ (*not sep-empty*))) $**$
(*not* (*sep-true* $\longrightarrow*$ (*not sep-empty*)))) *imp*
(*not* (*sep-true* $\longrightarrow*$ (*not sep-empty*))))
(*h*::*'a*::*heap-sep-algebra*)
by *separata*

The following is the axiom of indivisible unit. Sledgehammer finds a proof easily.

lemma *ax-iu*: ((*sep-empty and* (*A* $**$ *B*)) *imp A*)
(*h*::*'a*::*heap-sep-algebra*)
by *separata*

Sledgehammer fails to find a proof in 300s for this one.

lemma (*not* (((*A* $**$ (*C* $\longrightarrow*$ (*not* ((*not* (*A* $\longrightarrow*$ *B*)) $**$ *C*)))) *and* (*not B*)) $**$ *C*))
(*h*::*'a*::*heap-sep-algebra*)
by *separata*

Sledgehammer finds a proof easily.

lemma ((*sep-empty* $\longrightarrow*$ (*not* ((*not A*) $**$ *sep-empty*))) *imp A*)
(*h*::*'a*::*heap-sep-algebra*)
by *separata*

Sledgehammer finds a proof in 46 seconds.

lemma (*A imp* (*not* ((*not* (*A* $**$ *B*)) *and* (*not* (*A* $**$ (*not B*)))))))
(*h*::*'a*::*heap-sep-algebra*)
by *separata*

Sledgehammer easily finds a proof.

lemma ((*sep-empty and A*) *imp* (*A* $**$ *A*))
(*h*::*'a*::*heap-sep-algebra*)
by *separata*

Sledgehammer fails to find a proof in 300s.

lemma (*not* (((*A* ∗∗ (*C* −−∗ (*not* ((*not* (*A* −−∗ *B*)) ∗∗ *C*)))) *and* (*not B*)) ∗∗ *C*))
(*h*::′*a*::*heap-sep-algebra*)
by *separata*

Sledgehammer finds a proof easily.

lemma ((*sep-empty* −−∗ (*not* ((*not A*) ∗∗ *sep-empty*))) *imp A*)
(*h*::′*a*::*heap-sep-algebra*)
by *separata*

Sledgehammer finds a proof easily.

lemma (*sep-empty imp* ((*A* ∗∗ *B*) −−∗ (*B* ∗∗ *A*)))
(*h*::′*a*::*heap-sep-algebra*)
by *separata*

Sledgehammer takes a while to find a proof, although the proof is by smt and is fast.

lemma (*sep-empty imp* ((*A* ∗∗ (*B and C*)) −−∗ ((*A* ∗∗ *B*) *and* (*A* ∗∗ *C*))))
(*h*::′*a*::*heap-sep-algebra*)
by *separata*

Sledgehammer takes a long time to find a smt proof, but the smt proves it quickly.

lemma (*sep-empty imp* ((*A* −−∗ (*B imp C*)) −−∗ ((*A* −−∗ *B*) *imp* (*A* −−∗ *C*))))
(*h*::′*a*::*heap-sep-algebra*)
by *separata*

Sledgehammer finds a proof quickly.

lemma (*sep-empty imp* (((*A imp B*) −−∗ ((*A* −−∗ *A*) *imp A*)) *imp* (*A* −−∗ *A*)))
(*h*::′*a*::*heap-sep-algebra*)
by *separata*

Sledgehammer finds proofs in a while.

lemma ((*A* −−∗ *B*) *and* (*sep-true* ∗∗ (*sep-empty and A*)) *imp B*)
(*h*::′*a*::*heap-sep-algebra*)
by *separata*

Sledgehammer finds proofs easily.

lemma ((*sep-empty* −−∗ (*not* ((*not A*) ∗∗ *sep-true*))) *imp A*)
(*h*::′*a*::*heap-sep-algebra*)
by *separata*

Sledgehammer takes a while to find a proof.

lemma (*not* ((*A* −−∗ (*not* (*A* ∗∗ *B*))) *and* (((*not A*) −−∗ (*not B*)) *and B*)))

(*h*::*'a*::*heap-sep-algebra*)
by *separata*

Sledgehammer takes a long time to find a smt proof, although smt proves it quickly.

lemma (*sep-empty imp* ((*A* ⟶∗ (*B* ⟶∗ *C*)) ⟶∗ ((*A* ∗∗ *B*) ⟶∗ *C*)))
(*h*::*'a*::*heap-sep-algebra*)
by *separata*

Sledgehammer finds proofs easily.

lemma (*sep-empty imp* ((*A* ∗∗ (*B* ∗∗ *C*)) ⟶∗ ((*A* ∗∗ *B*) ∗∗ *C*)))
(*h*::*'a*::*heap-sep-algebra*)
by *separata*

Sledgehammer finds proofs in a few seconds.

lemma (*sep-empty imp* ((*A* ∗∗ ((*B* ⟶∗ *D*) ∗∗ *C*)) ⟶∗ ((*A* ∗∗ (*B* ⟶∗ *D*)) ∗∗ *C*)))
(*h*::*'a*::*heap-sep-algebra*)
by *separata*

Sledgehammer fails to find a proof in 300s.

lemma (*not* (((*A* ⟶∗ (*not* ((*not* (*D* ⟶∗ (*not* (*A* ∗∗ (*C* ∗∗ *B*))))) ∗∗ *A*))) *and* *C*) ∗∗ (*D and* (*A* ∗∗ *B*))))
(*h*::*'a*::*heap-sep-algebra*)
by *separata*

Sledgehammer takes a while to find a proof.

lemma (*not* ((*C* ∗∗ (*D* ∗∗ *E*)) *and* ((*A* ⟶∗ (*not* (*not* (*B* ⟶∗ *not* (*D* ∗∗ (*E* ∗∗ *C*))) ∗∗ *A*))) ∗∗
(*B and* (*A* ∗∗ *sep-true*)))))
(*h*::*'a*::*heap-sep-algebra*)
by *separata*

Sledgehammer fails to find a proof in 300s.

lemma (*not* (((*A* ⟶∗ (*not* ((*not* (*D* ⟶∗ (*not* ((*C* ∗∗ *E*) ∗∗ (*B* ∗∗ *A*))))) ∗∗ *A*))) *and* *C*) ∗∗ (*D and* (*A* ∗∗ (*B* ∗∗ *E*)))))
(*h*::*'a*::*heap-sep-algebra*)
by *separata*

Sledgehammer finds a proof easily.

lemma ((*A* ∗∗ (*B* ∗∗ (*C* ∗∗ (*D* ∗∗ *E*)))) *imp* (*E* ∗∗ (*B* ∗∗ (*A* ∗∗ (*C* ∗∗ *D*)))))
(*h*::*'a*::*heap-sep-algebra*)
by *separata*

lemma ((*A* ∗∗ (*B* ∗∗ (*C* ∗∗ (*D* ∗∗ (*E* ∗∗ (*F* ∗∗ *G*)))))) *imp* (*G* ∗∗ (*E* ∗∗ (*B* ∗∗ (*A* ∗∗ (*C* ∗∗ (*D* ∗∗ *F*)))))))
(*h*::*'a*::*heap-sep-algebra*)

by *separata*

Sledgehammer finds a proof in a few seconds.

lemma (*sep-empty imp* ((*A* ** ((*B* −→* *E*) ** (*C* ** *D*))) −→* ((*A* ** *D*) ** (*C* ** (*B* −→* *E*)))))
(*h*::*'a*::*heap-sep-algebra*)
by *separata*

This is the odd BBI formula that I personally can't prove using any other methods. I only know of a derivation in my labelled sequent calculus for BBI. Sledgehammer takes a while to find a proof.

lemma (*not* (*sep-empty and A and* (*B* ** (*not* (*C* −→* (*sep-empty imp A*))))))
(*h*::*'a*::*heap-sep-algebra*)
by *separata*

Sledgehammer finds a proof easily.

lemma ((((*sep-true imp p0*) *imp* ((*p0* ** *p0*) −→* ((*sep-true imp p0*) ** (*p0* ** *p0*)))) *imp*
(*p1* −→* (((*sep-true imp p0*) *imp* ((*p0* ** *p0*) −→* (((*sep-true imp p0*) ** *p0*) ** *p0*))) ** *p1*))))
(*h*::*'a*::*heap-sep-algebra*)
by *separata*

The following are some randomly generated BBI formulae.

Sledgehammer finds a proof easily.

lemma ((((*p1* −→* *p3*) −→* (*p5* −→* *p2*)) *imp* ((((*p7* ** *p4*) *and* (*p3* −→* *p2*)) *imp*
((*p7* ** *p4*) *and* (*p3* −→* *p2*))) −→* (((*p1* −→* *p3*) −→* (*p5* −→* *p2*)) **
(((*p4* ** *p7*) *and* (*p3* −→* *p2*)) *imp* ((*p4* ** *p7*) *and* (*p3* −→* *p2*)))))))
(*h*::*'a*::*heap-sep-algebra*)
by *separata*

Sledgehammer finds a proof easily.

lemma (((((*p1* −→* (*p0 imp sep-false*)) *imp sep-false*) *imp* (((*p1 imp sep-false*) *imp*
((*p0* ** ((*p1 imp sep-false*) −→* (*p4* −→* *p1*))) −→* ((*p1 imp sep-false*) **
(*p0* ** ((*p1 imp sep-false*) −→* (*p4* −→* *p1*)))))) *imp sep-false*)) *imp*

(((*p1 imp sep-false*) *imp* ((*p0* ** ((*p1 imp sep-false*) −→* (*p4* −→* *p1*))) −→*
((*p0* ** (*p1 imp sep-false*)) ** ((*p1 imp sep-false*) −→* (*p4* −→* *p1*)))))) *imp*
(*p1* −→* (*p0 imp sep-false*)))))
(*h*::*'a*::*heap-sep-algebra*)
by *separata*

Sledgehammer finds a proof easily.

lemma (((*p0 imp sep-false*) *imp* ((*p1* ** *p0*) −→* (*p1* ** ((*p0 imp sep-false*) **
p0)))) *imp* ((*p0 imp sep-false*) *imp* ((*p1* ** *p0*) −→* ((*p1* ** *p0*) ** (*p0 imp*
sep-false)))))
(*h*::′*a*::*heap-sep-algebra*)
by *separata*

Sledgehammer finds a proof in a while.

lemma (*sep-empty imp* ((((*p4* ** *p1*) −→* ((*p8* ** *sep-empty*) −→* *p0*)) *imp*
(*p1* −→* (*p1* ** ((*p4* ** *p1*) −→* ((*p8* ** *sep-empty*) −→* *p0*))))) −→*
(((*p4* ** *p1*) −→* ((*p8* ** *sep-empty*) −→* *p0*)) *imp* (*p1* −→* (((*p1* ** *p4*) −→*
((*p8* ** *sep-empty*) −→* *p0*)) ** *p1*)))))
(*h*::′*a*::*heap-sep-algebra*)
by *separata*

Sledgehammer finds a proof easily.

lemma ((((*p3 imp* (*p0* −→* (*p3* ** *p0*))) *imp sep-false*) *imp* (*p1 imp*
sep-false)) *imp*
(*p1 imp* (*p3 imp* (*p0* −→* (*p0* ** *p3*)))))
(*h*::′*a*::*heap-sep-algebra*)
by *separata*

Sledgehammer finds a proof in a few seconds.

lemma ((*p7* −→* (*p4* ** (*p6* −→* *p1*))) *imp* ((*p4 imp* (*p1* −→* ((*sep-empty* **
p1) ** *p4*))) −→* ((*p1 imp* (*p4* −→* (*p4* ** (*sep-empty* ** *p1*)))) ** (*p7* −→*
((*p6* −→* *p1*) ** *p4*)))))
(*h*::′*a*::*heap-sep-algebra*)
by *separata*

Sledgehammer finds a proof easily.

lemma (((*p2 imp p0*) *imp* ((*p0* ** *sep-true*) −→* (*p0* ** (*sep-true* **
(*p2 imp p0*))))) *imp* ((*p2 imp p0*) *imp* ((*sep-true* ** *p0*) −→*
(*p0* ** ((*p2 imp p0*) ** *sep-true*)))))
(*h*::′*a*::*heap-sep-algebra*)
by *separata*

Sledgehammer finds a proof easily.

lemma ((*sep-empty imp* ((*p1* −→* (((*p2 imp sep-false*) ** *p0*) ** *p8*)) −→*
(*p1* −→* ((*p2 imp sep-false*) ** (*p0* ** *p8*))))) *imp* ((*p0* ** *sep-empty*
) −→*

((*sep-empty* *imp* ((*p1* ⟶∗ ((*p0* ∗∗ (*p2 imp* *sep-false*)) ∗∗ *p8*)) ⟶∗ (*p1* ⟶∗
((*p2 imp* *sep-false*) ∗∗ (*p0* ∗∗ *p8*))))) ∗∗ (*p0* ∗∗ *sep-empty*))))
(*h*::′*a*::*heap-sep-algebra*)
by *separata*

Sledgehammer finds a proof in a while.

lemma ((*p0* ⟶∗ *sep-empty*) *imp* ((*sep-empty* *imp* ((*sep-empty* ∗∗ ((((*p8* ∗∗ *p7*) ∗∗
(*p8 imp* *p4*)) ⟶∗ *p8*) ∗∗ (*p2* ∗∗ *p1*))) ⟶∗ (*p2* ∗∗ (((*p7* ∗∗ ((*p8* *imp* *p4*) ∗∗
p8)) ⟶∗ *p8*) ∗∗ *p1*)))) ⟶∗ ((*sep-empty* *imp* (((((*p7* ∗∗ (*p8* ∗∗ (*p8* *imp* *p4*))) ⟶∗
p8) ∗∗ *sep-empty*) ∗∗ (*p1* ∗∗ *p2*)) ⟶∗ (((*p7* ∗∗ ((*p8 imp* *p4*) ∗∗ *p8*)) ⟶∗ *p8*) ∗∗
(*p1* ∗∗ *p2*)))) ∗∗ (*p0* ⟶∗ *sep-empty*))))
(*h*::′*a*::*heap-sep-algebra*)
by *separata*

end