Unbounded Separation Logic

Thibault Dardinier
Department of Computer Science, ETH Zurich, Switzerland

March 19, 2025

Abstract

Many separation logics [11] support fractional permissions [3, 2] to distinguish between read and write access to a heap location, for instance, to allow concurrent reads while enforcing exclusive writes. Fractional permissions extend to composite assertions such as (co)inductive predicates and magic wands by allowing those to be multiplied [8, 4, 6] by a fraction. Typical separation logic proofs require that this multiplication has three key properties: it needs to distribute over assertions, it should permit fractions to be factored out from assertions, and two fractions of the same assertion should be combinable into one larger fraction.

Existing formal semantics incorporating fractional assertions into a separation logic define multiplication semantically (via models), resulting in a semantics in which distributivity and combinability do not hold for key resource assertions such as magic wands, and fractions cannot be factored out from a separating conjunction. By contrast, existing automatic separation logic verifiers [9, 7, 10, 1] define multiplication syntactically, resulting in a different semantics for which it is unknown whether distributivity and combinability hold for all assertions.

In this entry, we present and formalize an *unbounded* version of separation logic [5], a novel semantics for separation logic assertions that allows states to hold more than a full permission to a heap location during the evaluation of an assertion. By reimposing upper bounds on the permissions held per location at statement boundaries, we retain key properties of separation logic, in particular, we prove that the frame rule still holds. We also prove that our assertion semantics unifies semantic and syntactic multiplication and thereby reconciles the discrepancy between separation logic theory and tools and enjoys distributivity, factorisability, and combinability.

Contents

1 Unbounded Separation Logic

theory *UnboundedLogic*
imports *Main*
begin

1.1 Assertions and state model

We define our assertion language as described in Section 2.3 of the paper [5].

datatype $('a, 'b, 'c, 'd)$ *assertion* =
Sem $('d \Rightarrow 'c) \Rightarrow 'a \Rightarrow$ *bool*
| *Mult* $'b$ $('a, 'b, 'c, 'd)$ *assertion*
| *Star* $('a, 'b, 'c, 'd)$ *assertion* $('a, 'b, 'c, 'd)$ *assertion*
| *Wand* $('a, 'b, 'c, 'd)$ *assertion* $('a, 'b, 'c, 'd)$ *assertion*
| *Or* $('a, 'b, 'c, 'd)$ *assertion* $('a, 'b, 'c, 'd)$ *assertion*
| *And* $('a, 'b, 'c, 'd)$ *assertion* $('a, 'b, 'c, 'd)$ *assertion*
| *Imp* $('a, 'b, 'c, 'd)$ *assertion* $('a, 'b, 'c, 'd)$ *assertion*
| *Exists* $'d$ $('a, 'b, 'c, 'd)$ *assertion*
| *Forall* $'d$ $('a, 'b, 'c, 'd)$ *assertion*
| *Pred*
| *Bounded* $('a, 'b, 'c, 'd)$ *assertion*
| *Wildcard* $('a, 'b, 'c, 'd)$ *assertion*

type-synonym $'a$ *command* = $('a \times 'a$ *option*$)$ *set*

locale *pre-logic* =
fixes *plus* :: $'a \Rightarrow 'a \Rightarrow 'a$ *option* (**infixl** ‹$\oplus$› *63*)

begin

definition *compatible* :: $'a \Rightarrow 'a \Rightarrow$ *bool* (**infixl** ‹##› *60*) **where**
a *##* $b \longleftrightarrow a \oplus b \neq$ *None*

definition *larger* :: $'a \Rightarrow 'a \Rightarrow$ *bool* (**infixl** ‹$\succeq$› *55*) **where**
$a \succeq b \longleftrightarrow (\exists\, c.$ *Some* $a = b \oplus c)$

end

type-synonym $('a, 'b, 'c)$ *interp* = $('a \Rightarrow 'b) \Rightarrow 'c$ *set*

The following locale captures the state model described in Section 2.2 of the paper [5].

locale *logic* = *pre-logic* +

fixes *mult* :: $'b \Rightarrow 'a \Rightarrow 'a$ (**infixl** ‹$\odot$› *64*)

fixes *smult* :: $'b \Rightarrow 'b \Rightarrow 'b$
fixes *sadd* :: $'b \Rightarrow 'b \Rightarrow 'b$
fixes *sinv* :: $'b \Rightarrow 'b$

fixes *one* :: *'b*

fixes *valid* :: *'a* ⇒ *bool*

assumes *commutative*: *a* ⊕ *b* = *b* ⊕ *a*
and *asso1*: *a* ⊕ *b* = *Some ab* ∧ *b* ⊕ *c* = *Some bc* ⟹ *ab* ⊕ *c* = *a* ⊕ *bc*
and *asso2*: *a* ⊕ *b* = *Some ab* ∧ ¬ *b* ## *c* ⟹ ¬ *ab* ## *c*

and *sinv-inverse*: *smult p* (*sinv p*) = *one*
and *sone-neutral*: *smult one p* = *p*
and *sadd-comm*: *sadd p q* = *sadd q p*
and *smult-comm*: *smult p q* = *smult q p*
and *smult-distrib*: *smult p* (*sadd q r*) = *sadd* (*smult p q*) (*smult p r*)
and *smult-asso*: *smult* (*smult p q*) *r* = *smult p* (*smult q r*)

and *double-mult*: *p* ⊙ (*q* ⊙ *a*) = (*smult p q*) ⊙ *a*
and *plus-mult*: *Some a* = *b* ⊕ *c* ⟹ *Some* (*p* ⊙ *a*) = (*p* ⊙ *b*) ⊕ (*p* ⊙ *c*)
and *distrib-mult*: *Some* ((*sadd p q*) ⊙ *x*) = *p* ⊙ *x* ⊕ *q* ⊙ *x*
and *one-neutral*: *one* ⊙ *a* = *a*

and *valid-mono*: *valid a* ∧ *a* ≽ *b* ⟹ *valid b*

begin

The validity of assertions corresponds to Figure 3 of the paper [5].

fun *sat* :: *'a* ⇒ (*'d* ⇒ *'c*) ⇒ (*'d*, *'c*, *'a*) *interp* ⇒ (*'a*, *'b*, *'c*, *'d*) *assertion* ⇒ *bool* (‹-, -, - ⊨ -› [*51*, *65*, *68*, *66*] *50*) **where**
σ, *s*, Δ ⊨ *Mult p A* ⟷ (∃ *a*. *σ* = *p* ⊙ *a* ∧ *a*, *s*, Δ ⊨ *A*)
| *σ*, *s*, Δ ⊨ *Star A B* ⟷ (∃ *a b*. *Some σ* = *a* ⊕ *b* ∧ *a*, *s*, Δ ⊨ *A* ∧ *b*, *s*, Δ ⊨ *B*)
| *σ*, *s*, Δ ⊨ *Wand A B* ⟷ (∀ *a σ'*. *a*, *s*, Δ ⊨ *A* ∧ *Some σ'* = *σ* ⊕ *a* ⟶ *σ'*, *s*, Δ ⊨ *B*)

| *σ*, *s*, Δ ⊨ *Sem b* ⟷ *b s σ*
| *σ*, *s*, Δ ⊨ *Imp A B* ⟷ (*σ*, *s*, Δ ⊨ *A* ⟶ *σ*, *s*, Δ ⊨ *B*)
| *σ*, *s*, Δ ⊨ *Or A B* ⟷ (*σ*, *s*, Δ ⊨ *A* ∨ *σ*, *s*, Δ ⊨ *B*)
| *σ*, *s*, Δ ⊨ *And A B* ⟷ (*σ*, *s*, Δ ⊨ *A* ∧ *σ*, *s*, Δ ⊨ *B*)

| *σ*, *s*, Δ ⊨ *Exists x A* ⟷ (∃ *v*. *σ*, *s*(*x* := *v*), Δ ⊨ *A*)
| *σ*, *s*, Δ ⊨ *Forall x A* ⟷ (∀ *v*. *σ*, *s*(*x* := *v*), Δ ⊨ *A*)

| *σ*, *s*, Δ ⊨ *Pred* ⟷ (*σ* ∈ Δ *s*)
| *σ*, *s*, Δ ⊨ *Bounded A* ⟷ (*valid σ* ⟶ *σ*, *s*, Δ ⊨ *A*)
| *σ*, *s*, Δ ⊨ *Wildcard A* ⟷ (∃ *a p*. *σ* = *p* ⊙ *a* ∧ *a*, *s*, Δ ⊨ *A*)

definition *intuitionistic* :: (*'d* ⇒ *'c*) ⇒ (*'d*, *'c*, *'a*) *interp* ⇒ (*'a*, *'b*, *'c*, *'d*) *assertion* ⇒ *bool* **where**
intuitionistic s Δ *A* ⟷ (∀ *a b*. *a* ≽ *b* ∧ *b*, *s*, Δ ⊨ *A* ⟶ *a*, *s*, Δ ⊨ *A*)

definition *entails* :: ('*a*, '*b*, '*c*, '*d*) *assertion* ⇒ ('*d*, '*c*, '*a*) *interp* ⇒ ('*a*, '*b*, '*c*, '*d*) *assertion* ⇒ *bool* (‹-, - ⊢ -› [*63*, *66*, *68*] *52*) **where**
A, Δ ⊢ *B* ⟷ (∀ σ *s*. σ, *s*, Δ ⊨ *A* ⟶ σ, *s*, Δ ⊨ *B*)

definition *equivalent* :: ('*a*, '*b*, '*c*, '*d*) *assertion* ⇒ ('*d*, '*c*, '*a*) *interp* ⇒ ('*a*, '*b*, '*c*, '*d*) *assertion* ⇒ *bool* (‹-, - ≡ -› [*63*, *66*, *68*] *52*) **where**
A, Δ ≡ *B* ⟷ (*A*, Δ ⊢ *B* ∧ *B*, Δ ⊢ *A*)

definition *pure* :: ('*a*, '*b*, '*c*, '*d*) *assertion* ⇒ *bool* **where**
pure A ⟷ (∀ σ σ′ *s* Δ Δ′. σ, *s*, Δ ⊨ *A* ⟷ σ′, *s*, Δ′ ⊨ *A*)

1.2 Useful lemmas

lemma *sat-forall*:
assumes ⋀*v*. σ, *s*(*x* := *v*), Δ ⊨ *A*
shows σ, *s*, Δ ⊨ *Forall x A*
by (*simp add*: *assms*)

lemma *intuitionisticI*:
assumes ⋀*a b*. *a* ≽ *b* ∧ *b*, *s*, Δ ⊨ *A* ⟹ *a*, *s*, Δ ⊨ *A*
shows *intuitionistic s* Δ *A*
by (*meson assms intuitionistic-def*)

lemma *can-divide*:
assumes *p* ⊙ *a* = *p* ⊙ *b*
shows *a* = *b*
by (*metis assms double-mult logic.one-neutral logic-axioms sinv-inverse smult-comm*)

lemma *unique-inv*:
a = *p* ⊙ *b* ⟷ *b* = (*sinv p*) ⊙ *a*
by (*metis double-mult logic.can-divide logic-axioms sinv-inverse sone-neutral*)

lemma *entailsI*:
assumes ⋀σ *s*. σ, *s*, Δ ⊨ *A* ⟹ σ, *s*, Δ ⊨ *B*
shows *A*, Δ ⊢ *B*
by (*simp add*: *assms entails-def*)

lemma *equivalentI*:
assumes ⋀σ *s*. σ, *s*, Δ ⊨ *A* ⟹ σ, *s*, Δ ⊨ *B*
and ⋀σ *s*. σ, *s*, Δ ⊨ *B* ⟹ σ, *s*, Δ ⊨ *A*
shows *A*, Δ ≡ *B*
by (*simp add*: *assms*(*1*) *assms*(*2*) *entailsI equivalent-def*)

lemma *compatible-imp*:
assumes *a* ## *b*
shows (*p* ⊙ *a*) ## (*p* ⊙ *b*)
by (*metis assms compatible-def option.distinct*(*1*) *option.exhaust plus-mult*)

lemma *compatible-iff*:
  $a \mathbin{\#\#} b \longleftrightarrow (p \odot a) \mathbin{\#\#} (p \odot b)$
  by (*metis compatible-imp unique-inv*)

lemma *sat-wand*:
  assumes $\bigwedge a\ \sigma'.\ a,\ s,\ \Delta \models A \wedge \mathit{Some}\ \sigma' = \sigma \oplus a \Longrightarrow \sigma',\ s,\ \Delta \models B$
  shows $\sigma,\ s,\ \Delta \models \mathit{Wand}\ A\ B$
  using *assms* **by** *auto*

lemma *sat-imp*:
  assumes $\sigma,\ s,\ \Delta \models A \Longrightarrow \sigma,\ s,\ \Delta \models B$
  shows $\sigma,\ s,\ \Delta \models \mathit{Imp}\ A\ B$
  using *assms* **by** *auto*

lemma *sat-mult*:
  assumes $\bigwedge a.\ \sigma = p \odot a \Longrightarrow a,\ s,\ \Delta \models A$
  shows $\sigma,\ s,\ \Delta \models \mathit{Mult}\ p\ A$
  by (*metis assms logic.sat.simps*(*1*) *logic-axioms unique-inv*)

lemma *larger-same*:
  $a \succeq b \longleftrightarrow p \odot a \succeq p \odot b$
proof −
  have $\bigwedge a\ b\ p.\ a \succeq b \Longrightarrow p \odot a \succeq p \odot b$
    by (*meson larger-def plus-mult*)
  then show *?thesis*
    by (*metis unique-inv*)
qed

lemma *asso3*:
  assumes $\neg\ a \mathbin{\#\#} b$
      and $b \oplus c = \mathit{Some}\ bc$
    shows $\neg\ a \mathbin{\#\#} bc$
  by (*metis* (*full-types*) *assms*(*1*) *assms*(*2*) *asso2 commutative compatible-def*)

lemma *compatible-smaller*:
  assumes $a \succeq b$
      and $x \mathbin{\#\#} a$
    shows $x \mathbin{\#\#} b$
  by (*metis assms*(*1*) *assms*(*2*) *asso3 larger-def*)

lemma *compatible-multiples*:
  assumes $p \odot a \mathbin{\#\#} q \odot b$
  shows $a \mathbin{\#\#} b$
  by (*metis* (*no-types*, *opaque-lifting*) *assms commutative compatible-def compatible-iff compatible-smaller distrib-mult larger-def one-neutral*)

lemma *move-sum*:
  assumes $\mathit{Some}\ a = a1 \oplus a2$
      and $\mathit{Some}\ b = b1 \oplus b2$

and *Some x = a ⊕ b*
and *Some x1 = a1 ⊕ b1*
and *Some x2 = a2 ⊕ b2*
shows *Some x = x1 ⊕ x2*
proof −
obtain *ab1* **where** *Some ab1 = a ⊕ b1*
by (*metis assms*(2) *assms*(3) *asso3 compatible-def not-Some-eq*)
then have *Some ab1 = x1 ⊕ a2*
by (*metis assms*(1) *assms*(4) *asso1 commutative*)
then show *?thesis*
by (*metis ‹Some ab1 = a ⊕ b1› assms*(2) *assms*(3) *assms*(5) *asso1*)
qed

lemma *sum-both-larger*:
assumes *Some x′ = a′ ⊕ b′*
and *Some x = a ⊕ b*
and *a′ ≽ a*
and *b′ ≽ b*
shows *x′ ≽ x*
proof −
obtain *ra rb* **where** *Some a′ = a ⊕ ra Some b′ = b ⊕ rb*
using *assms*(3) *assms*(4) *larger-def* **by** *auto*
then obtain *r* **where** *Some r = ra ⊕ rb*
by (*metis assms*(1) *asso3 commutative compatible-def option.collapse*)
then have *Some x′ = x ⊕ r*
by (*meson ‹Some a′ = a ⊕ ra› ‹Some b′ = b ⊕ rb› assms*(1) *assms*(2) *move-sum*)
then show *?thesis*
using *larger-def* **by** *blast*
qed

lemma *larger-first-sum*:
assumes *Some y = a ⊕ b*
and *x ≽ y*
shows ∃ *a′*. *Some x = a′ ⊕ b ∧ a′ ≽ a*
proof −
obtain *r* **where** *Some x = y ⊕ r*
using *assms*(2) *larger-def* **by** *auto*
then obtain *a′* **where** *Some a′ = a ⊕ r*
by (*metis assms*(1) *asso2 commutative compatible-def option.collapse*)
then show *?thesis*
by (*metis ‹Some x = y ⊕ r› assms*(1) *asso1 commutative larger-def*)
qed

lemma *larger-implies-compatible*:
assumes *x ≽ y*
shows *x ## y*
by (*metis assms compatible-def compatible-smaller distrib-mult one-neutral option.distinct*(1))

2 Frame rule

This section corresponds to Section 2.5 of the paper [5].

definition *safe* :: $('a \times ('d \Rightarrow 'c))$ *command* $\Rightarrow ('a \times ('d \Rightarrow 'c)) \Rightarrow$ *bool* **where**
safe c σ $\longleftrightarrow (\sigma, None) \notin c$

definition *safety-monotonicity* :: $('a \times ('d \Rightarrow 'c))$ *command* $\Rightarrow$ *bool* **where**
safety-monotonicity c $\longleftrightarrow (\forall \sigma\ \sigma'\ s.\ valid\ \sigma' \wedge \sigma' \succeq \sigma \wedge safe\ c\ (\sigma, s) \longrightarrow safe\ c\ (\sigma', s))$

definition *frame-property* :: $('a \times ('d \Rightarrow 'c))$ *command* $\Rightarrow$ *bool* **where**
frame-property c $\longleftrightarrow (\forall \sigma\ \sigma 0\ r\ \sigma'\ s\ s'.\ valid\ \sigma \wedge valid\ \sigma' \wedge safe\ c\ (\sigma 0, s) \wedge Some\ \sigma = \sigma 0 \oplus r \wedge ((\sigma, s), Some\ (\sigma', s')) \in c$
$\longrightarrow (\exists \sigma 0'.\ Some\ \sigma' = \sigma 0' \oplus r \wedge ((\sigma 0, s), Some\ (\sigma 0', s')) \in c))$

definition *valid-hoare-triple* :: $('a, 'b, 'c, 'd)$ *assertion* $\Rightarrow ('a \times ('d \Rightarrow 'c))$ *command* $\Rightarrow ('a, 'b, 'c, 'd)$ *assertion* $\Rightarrow ('d, 'c, 'a)$ *interp* $\Rightarrow$ *bool* **where**
valid-hoare-triple P c Q Δ $\longleftrightarrow (\forall \sigma\ s.\ valid\ \sigma \wedge \sigma, s, \Delta \models P \longrightarrow safe\ c\ (\sigma, s) \wedge (\forall \sigma'\ s'.\ ((\sigma, s), Some\ (\sigma', s')) \in c \longrightarrow \sigma', s', \Delta \models Q))$

lemma *valid-hoare-tripleI*:
assumes $\bigwedge \sigma\ s.\ valid\ \sigma \wedge \sigma, s, \Delta \models P \Longrightarrow safe\ c\ (\sigma, s)$
and $\bigwedge \sigma\ s\ \sigma'\ s'.\ valid\ \sigma \wedge \sigma, s, \Delta \models P \Longrightarrow ((\sigma, s), Some\ (\sigma', s')) \in c \Longrightarrow \sigma', s', \Delta \models Q$
shows *valid-hoare-triple P c Q Δ*
using *assms(1) assms(2) valid-hoare-triple-def* **by** *blast*

definition *valid-command* :: $('a \times ('d \Rightarrow 'c))$ *command* $\Rightarrow$ *bool* **where**
valid-command c $\longleftrightarrow (\forall a\ b\ sa\ sb.\ ((a, sa), Some\ (b, sb)) \in c \wedge valid\ a \longrightarrow valid\ b)$

definition *modified* :: $('a \times ('d \Rightarrow 'c))$ *command* $\Rightarrow 'd$ *set* **where**
modified c $= \{\ x\ |x.\ \exists \sigma\ s\ \sigma'\ s'.\ ((\sigma, s), Some\ (\sigma', s')) \in c \wedge s\ x \neq s'\ x\ \}$

definition *equal-outside* :: $('d \Rightarrow 'c) \Rightarrow ('d \Rightarrow 'c) \Rightarrow 'd$ *set* $\Rightarrow$ *bool* **where**
equal-outside s s' S $\longleftrightarrow (\forall x.\ x \notin S \longrightarrow s\ x = s'\ x)$

definition *not-in-fv* :: $('a, 'b, 'c, 'd)$ *assertion* $\Rightarrow 'd$ *set* $\Rightarrow$ *bool* **where**
not-in-fv A S $\longleftrightarrow (\forall \sigma\ s\ \Delta\ s'.\ equal\text{-}outside\ s\ s'\ S \longrightarrow (\sigma, s, \Delta \models A \longleftrightarrow \sigma, s', \Delta \models A))$

lemma *not-in-fv-mod*:
assumes *not-in-fv A* (*modified c*)
and $((\sigma, s), Some\ (\sigma', s')) \in c$
shows $x, s, \Delta \models A \longleftrightarrow x, s', \Delta \models A$

proof −
have $\bigwedge x.\ x \notin (\mathit{modified}\ c) \Longrightarrow s\ x = s'\ x$
proof −
fix x **assume** $x \notin (\mathit{modified}\ c)$
then show $s\ x = s'\ x$
by (*metis* (*mono-tags*, *lifting*) *CollectI assms*(2) *modified-def*)
qed
then have *equal-outside* $s\ s'$ (*modified* c)
by (*simp add*: *equal-outside-def*)
then show *?thesis*
using *assms*(1) *not-in-fv-def* **by** *blast*
qed

This theorem corresponds to Theorem 2 of the paper [5].

theorem *frame-rule*:
assumes *valid-command* c
and *safety-monotonicity* c
and *frame-property* c
and *valid-hoare-triple* $P\ c\ Q\ \Delta$
and *not-in-fv* R (*modified* c)
shows *valid-hoare-triple* (*Star* $P\ R$) c (*Star* $Q\ R$) Δ
proof (*rule valid-hoare-tripleI*)
fix $\sigma\ s$ **assume** *asm0*: *valid* $\sigma \wedge \sigma,\ s,\ \Delta \models \mathit{Star}\ P\ R$
then obtain $p\ r$ **where** *Some* $\sigma = p \oplus r\ p,\ s,\ \Delta \models P\ r,\ s,\ \Delta \models R$
by *auto*
then have *safe* c $(p,\ s)$
by (*metis asm0 assms*(4) *larger-def logic.valid-mono logic-axioms valid-hoare-triple-def*)
then show *safe* c $(\sigma,\ s)$
using ‹*Some* $\sigma = p \oplus r$› *assms*(2) *larger-def safety-monotonicity-def asm0* **by**
blast
fix $\sigma'\ s'$ **assume** *asm1*: $((\sigma,\ s),\ \mathit{Some}\ (\sigma',\ s')) \in c$
then obtain q **where** *Some* $\sigma' = q \oplus r\ ((p,\ s),\ \mathit{Some}\ (q,\ s')) \in c$
using ‹*Some* $\sigma = p \oplus r$› ‹*safe* c $(p,\ s)$› *asm0 assms*(1) *assms*(3) *frame-property-def*
valid-command-def **by** *blast*
moreover have $r,\ s',\ \Delta \models R$
by (*meson* ‹$r,\ s,\ \Delta \models R$› *assms*(5) *calculation*(2) *logic.not-in-fv-mod logic-axioms*)
ultimately show $\sigma',\ s',\ \Delta \models \mathit{Star}\ Q\ R$
by (*meson* ‹*Some* $\sigma = p \oplus r$› ‹$p,\ s,\ \Delta \models P$› ‹$r,\ s,\ \Delta \models R$› *asm0 assms*(4)
larger-def sat.simps(2) *valid-hoare-triple-def valid-mono*)
qed

lemma *hoare-triple-input*:
valid-hoare-triple $P\ c\ Q\ \Delta \longleftrightarrow$ *valid-hoare-triple* (*Bounded* P) $c\ Q\ \Delta$
using *sat.simps*(11) *valid-hoare-triple-def* **by** *blast*

lemma *hoare-triple-output*:
assumes *valid-command* c

shows *valid-hoare-triple P c Q* Δ ⟷ *valid-hoare-triple P c* (*Bounded Q*) Δ
using *assms valid-command-def valid-hoare-triple-def* **by** *fastforce*

end

end

3 Distributivity and Factorisability

This section corresponds to Section 2.4 and Figure 4 of the paper [5].

theory *Distributivity*
imports *UnboundedLogic*
begin

context *logic*
begin

3.1 DotPos

lemma *DotPos*:
A, Δ ⊢ *B* ⟷ (*Mult π A*, Δ ⊢ *Mult π B*) (**is** *?A* ⟷ *?B*)
proof
show *?A* ⟹ *?B*
by (*metis* (*no-types*, *lifting*) *entails-def sat.simps*(*1*))
show *?B* ⟹ *?A*
using *can-divide entails-def sat.simps*(*1*)
by *metis*
qed

Only one direction holds with a wildcard

lemma *WildPos*:
A, Δ ⊢ *B* ⟹ (*Wildcard A*, Δ ⊢ *Wildcard B*)
by (*metis* (*no-types*, *lifting*) *entails-def sat.simps*(*12*))

3.2 DotDot

lemma *dot-mult1*:
Mult p (*Mult q A*), Δ ⊢ *Mult* (*smult p q*) *A*
proof (*rule entailsI*)
fix *σ s*
assume *σ*, *s*, Δ ⊨ *Mult p* (*Mult q A*)
then show *σ*, *s*, Δ ⊨ *Mult* (*smult p q*) *A*
using *double-mult* **by** *auto*
qed

lemma *dot-mult2*:

```
  Mult (smult p q) A, Δ ⊢ Mult p (Mult q A)
proof (rule entailsI)
  fix σ s Δ
  assume σ, s, Δ ⊨ Mult (smult p q) A
  then obtain a where a, s, Δ ⊨ A σ = (smult p q) ⊙ a
    by auto
  then have q ⊙ a, s, Δ ⊨ Mult q A by auto
  then show σ, s, Δ ⊨ Mult p (Mult q A)
    by (metis ‹σ = smult p q ⊙ a› double-mult sat.simps(1))
qed

lemma DotDot:
  Mult p (Mult q A), Δ ≡ Mult (smult p q) A
  by (simp add: dot-mult1 dot-mult2 equivalent-def)

lemma can-factorize:
  ∃ r. q = smult r p
  by (metis sinv-inverse smult-asso smult-comm sone-neutral)

lemma WildDot:
  Wildcard (Mult p A), Δ ≡ Wildcard A
proof (rule equivalentI)
  show ⋀σ s. σ, s, Δ ⊨ Wildcard (Mult p A) ⟹ σ, s, Δ ⊨ Wildcard A
    using double-mult by fastforce
  fix σ s
  assume asm0: σ, s, Δ ⊨ Wildcard A
  then obtain q a where σ = q ⊙ a a, s, Δ ⊨ A
    using sat.simps(12) by blast
  then obtain r where q = smult r p
    using can-factorize by blast
  then have σ = r ⊙ (p ⊙ a)
    by (simp add: ‹σ = q ⊙ a› double-mult)
  then show σ, s, Δ ⊨ Wildcard (Mult p A)
    using ‹a, s, Δ ⊨ A› sat.simps(1) sat.simps(12) by blast
qed

lemma DotWild:
  Mult p (Wildcard A), Δ ≡ Wildcard A
proof (rule equivalentI)
  show ⋀σ s. σ, s, Δ ⊨ Mult p (Wildcard A) ⟹ σ, s, Δ ⊨ Wildcard A
    using double-mult by fastforce
  fix σ s
  assume asm0: σ, s, Δ ⊨ Wildcard A
  then obtain q a where σ = q ⊙ a a, s, Δ ⊨ A
    by force
  then obtain r where q = smult p r
    using can-factorize smult-comm by presburger
  then have σ = p ⊙ (r ⊙ a)
    by (simp add: ‹σ = q ⊙ a› double-mult)
```

then show σ, s, $\Delta \models$ *Mult p* (*Wildcard A*)
using ‹a, s, $\Delta \models A$› **by** *auto*
qed

lemma *WildWild*:
Wildcard (*Wildcard A*), $\Delta \equiv$ *Wildcard A*
proof (*rule equivalentI*)
show $\bigwedge\sigma$ s. σ, s, $\Delta \models$ *Wildcard* (*Wildcard A*) $\Longrightarrow$ σ, s, $\Delta \models$ *Wildcard A*
using *double-mult* **by** *fastforce*
show $\bigwedge\sigma$ s. σ, s, $\Delta \models$ *Wildcard A* $\Longrightarrow$ σ, s, $\Delta \models$ *Wildcard* (*Wildcard A*)
by (*metis one-neutral sat.simps*(*12*))
qed

3.3 DotStar

lemma *dot-star1*:
Mult p (*Star A B*), $\Delta \vdash$ *Star* (*Mult p A*) (*Mult p B*)
proof (*rule entailsI*)
fix σ s Δ
assume σ, s, $\Delta \models$ *Mult p* (*Star A B*)
then obtain a b x **where** $\sigma = p \odot x$ *Some* $x = a \oplus b$ a, s, $\Delta \models A$ b, s, $\Delta \models B$
by *auto*
then show σ, s, $\Delta \models$ *Star* (*Mult p A*) (*Mult p B*)
using *plus-mult* **by** *auto*
qed

lemma *dot-star2*:
Star (*Mult p A*) (*Mult p B*), $\Delta \vdash$ *Mult p* (*Star A B*)
proof (*rule entailsI*)
fix σ s Δ
assume σ, s, $\Delta \models$ *Star* (*Mult p A*) (*Mult p B*)
then obtain a b **where** *Some* $\sigma = (p \odot a) \oplus (p \odot b)$ a, s, $\Delta \models A$ b, s, $\Delta \models B$
by *auto*
then obtain x **where** *Some* $x = a \oplus b$
by (*metis plus-mult unique-inv*)
then have $\sigma = p \odot x$
by (*metis* ‹*Some* $\sigma = p \odot a \oplus p \odot b$› *option.sel plus-mult*)
then show σ, s, $\Delta \models$ *Mult p* (*Star A B*)
using ‹*Some* $x = a \oplus b$› ‹a, s, $\Delta \models A$› ‹b, s, $\Delta \models B$› **by** *auto*
qed

lemma *DotStar*:
Mult p (*Star A B*), $\Delta \equiv$ *Star* (*Mult p A*) (*Mult p B*)
by (*simp add*: *dot-star1 dot-star2 equivalent-def*)

lemma *WildStar1*:
Wildcard (*Star A B*), $\Delta \vdash$ *Star* (*Wildcard A*) (*Wildcard B*)
proof (*rule entailsI*)

fix σ s **assume** *asm0*: σ, s, $\Delta \models$ *Wildcard* (*Star A B*)
then obtain p *ab* a b **where** $\sigma = p \odot ab$ *Some* $ab = a \oplus b$ a, s, $\Delta \models A$ b, s, $\Delta \models B$
by *auto*
then have *Some* $\sigma = (p \odot a) \oplus (p \odot b)$
using *plus-mult* **by** *blast*
then show σ, s, $\Delta \models$ *Star* (*Wildcard A*) (*Wildcard B*)
using ‹a, s, $\Delta \models A$› ‹b, s, $\Delta \models B$› **by** *auto*
qed

3.4 DotWand

lemma *dot-wand1*:
Mult p (*Wand A B*), $\Delta \vdash$ *Wand* (*Mult p A*) (*Mult p B*)
proof (*rule entailsI*)
fix σ s Δ
assume σ, s, $\Delta \models$ *Mult p* (*Wand A B*)
then obtain x **where** $\sigma = p \odot x$ x, s, $\Delta \models$ *Wand A B*
by *auto*
show σ, s, $\Delta \models$ *Wand* (*Mult p A*) (*Mult p B*)
proof (*rule sat-wand*)
fix a σ'
assume a, s, $\Delta \models$ *Mult p A* $\wedge$ *Some* $\sigma' = \sigma \oplus a$
then obtain *aa* **where** *aa*, s, $\Delta \models A$ $a = p \odot aa$
by *auto*
then obtain b **where** *Some* $b = x \oplus aa$
by (*metis* ‹$\sigma = p \odot x$› ‹a, s, $\Delta \models$ *Mult p A* $\wedge$ *Some* $\sigma' = \sigma \oplus a$› *compatible-def compatible-iff option.exhaust-sel*)
then have b, s, $\Delta \models B$
using ‹*aa*, s, $\Delta \models A$› ‹x, s, $\Delta \models$ *Wand A B*› **by** *auto*
then show σ', s, $\Delta \models$ *Mult p B*
by (*metis* ‹*Some* $b = x \oplus aa$› ‹$\sigma = p \odot x$› ‹$a = p \odot aa$› ‹a, s, $\Delta \models$ *Mult p A* $\wedge$ *Some* $\sigma' = \sigma \oplus a$› *can-divide option.inject plus-mult sat-mult*)
qed
qed

lemma *dot-wand2*:
Wand (*Mult p A*) (*Mult p B*), $\Delta \vdash$ *Mult p* (*Wand A B*)
proof (*rule entailsI*)
fix σ s Δ
assume *asm*: σ, s, $\Delta \models$ *Wand* (*Mult p A*) (*Mult p B*)
show σ, s, $\Delta \models$ *Mult p* (*Wand A B*)
proof (*rule sat-mult*)
fix a **assume** $\sigma = p \odot a$
show a, s, $\Delta \models$ *Wand A B*
proof (*rule sat-wand*)
fix *aa* σ'
assume *aa*, s, $\Delta \models A$ $\wedge$ *Some* $\sigma' = a \oplus aa$
then have $p \odot aa$, s, $\Delta \models$ *Mult p A* **by** *auto*

then have *Some* $(p \odot \sigma') = \sigma \oplus p \odot aa$
by (*simp add*: ‹$\sigma = p \odot a$› ‹aa, s, $\Delta \models A \wedge$ *Some* $\sigma' = a \oplus aa$› *plus-mult*)
then have $p \odot \sigma'$, s, $\Delta \models$ *Mult* p B
using ‹$p \odot aa$, s, $\Delta \models$ *Mult* p A› *asm* **by** *force*
then show σ', s, $\Delta \models B$
by (*metis can-divide sat.simps*(*1*))
qed
qed
qed

lemma *DotWand*:
Mult p (*Wand* A B), $\Delta \equiv$ *Wand* (*Mult* p A) (*Mult* p B)
by (*simp add*: *dot-wand1 dot-wand2 equivalent-def*)

3.5 DotOr

lemma *dot-or1*:
Mult p (*Or* A B), $\Delta \vdash$ *Or* (*Mult* p A) (*Mult* p B)
proof (*rule entailsI*)
fix σ s Δ
assume σ, s, $\Delta \models$ *Mult* p (*Or* A B)
then obtain x **where** $\sigma = p \odot x$ x, s, $\Delta \models A \vee x$, s, $\Delta \models B$
by *auto*
then show σ, s, $\Delta \models$ *Or* (*Mult* p A) (*Mult* p B)
proof (*cases* x, s, $\Delta \models A$)
case *True*
then show *?thesis*
using ‹$\sigma = p \odot x$› **by** *auto*
next
case *False*
then show *?thesis*
using ‹$\sigma = p \odot x$› ‹x, s, $\Delta \models A \vee x$, s, $\Delta \models B$› **by** *auto*
qed
qed

lemma *dot-or2*:
Or (*Mult* p A) (*Mult* p B), $\Delta \vdash$ *Mult* p (*Or* A B)
proof (*rule entailsI*)
fix σ s Δ
assume σ, s, $\Delta \models$ *Or* (*Mult* p A) (*Mult* p B)
then show σ, s, $\Delta \models$ *Mult* p (*Or* A B)
proof (*cases* σ, s, $\Delta \models$ *Mult* p A)
case *True*
then show *?thesis* **by** *auto*
next
case *False*
then show *?thesis*
using ‹σ, s, $\Delta \models$ *Or* (*Mult* p A) (*Mult* p B)› **by** *auto*
qed

qed

lemma *DotOr*:
Mult p (*Or A B*), Δ ≡ *Or* (*Mult p A*) (*Mult p B*)
by (*simp add*: *dot-or1 dot-or2 equivalent-def*)

lemma *WildOr*:
Wildcard (*Or A B*), Δ ≡ *Or* (*Wildcard A*) (*Wildcard B*)
proof (*rule equivalentI*)
show ⋀σ *s*. σ, *s*, Δ ⊨ *Wildcard* (*Or A B*) ⟹ σ, *s*, Δ ⊨ *Or* (*Wildcard A*) (*Wildcard B*)
by *auto*
show ⋀σ *s*. σ, *s*, Δ ⊨ *Or* (*Wildcard A*) (*Wildcard B*) ⟹ σ, *s*, Δ ⊨ *Wildcard* (*Or A B*)
by *auto*
qed

3.6 DotAnd

lemma *dot-and1*:
Mult p (*And A B*), Δ ⊢ *And* (*Mult p A*) (*Mult p B*)
proof (*rule entailsI*)
fix σ *s* Δ
assume σ, *s*, Δ ⊨ *Mult p* (*And A B*)
then obtain *x* **where** σ = *p* ⊙ *x* *x*, *s*, Δ ⊨ *A* *x*, *s*, Δ ⊨ *B*
by *auto*
then show σ, *s*, Δ ⊨ *And* (*Mult p A*) (*Mult p B*)
by *auto*
qed

lemma *dot-and2*:
And (*Mult p A*) (*Mult p B*), Δ ⊢ *Mult p* (*And A B*)
proof (*rule entailsI*)
fix σ *s* Δ
assume σ, *s*, Δ ⊨ *And* (*Mult p A*) (*Mult p B*)
then show σ, *s*, Δ ⊨ *Mult p* (*And A B*)
using *logic.can-divide logic-axioms* **by** *fastforce*
qed

lemma *DotAnd*:
And (*Mult p A*) (*Mult p B*), Δ ≡ *Mult p* (*And A B*)
by (*simp add*: *dot-and1 dot-and2 equivalent-def*)

lemma *WildAnd*:
Wildcard (*And A B*), Δ ⊢ *And* (*Wildcard A*) (*Wildcard B*)
using *entails-def* **by** *fastforce*

3.7 DotImp

lemma *dot-imp1*:

Imp (*Mult p A*) (*Mult p B*), Δ ⊢ *Mult p* (*Imp A B*)
proof (*rule entailsI*)
fix σ *s* Δ
assume σ, *s*, Δ ⊨ *Imp* (*Mult p A*) (*Mult p B*)
then show σ, *s*, Δ ⊨ *Mult p* (*Imp A B*)
using *sat-mult* **by** *force*
qed

lemma *dot-imp2*:
Mult p (*Imp A B*), Δ ⊢ *Imp* (*Mult p A*) (*Mult p B*)
proof (*rule entailsI*)
fix σ *s* Δ
assume σ, *s*, Δ ⊨ *Mult p* (*Imp A B*)
then show σ, *s*, Δ ⊨ *Imp* (*Mult p A*) (*Mult p B*)
using *can-divide* **by** *auto*
qed

lemma *DotImp*:
Mult p (*Imp A B*), Δ ≡ *Imp* (*Mult p A*) (*Mult p B*)
by (*simp add*: *dot-imp1 dot-imp2 equivalent-def*)

3.8 DotPure

lemma *pure-mult1*:
assumes *pure A*
shows *Mult p A*, Δ ⊢ *A*
using *assms entails-def logic.pure-def logic-axioms* **by** *fastforce*

lemma *pure-mult2*:
assumes *pure A*
shows *A*, Δ ⊢ *Mult p A*
using *assms entailsI pure-def sat-mult*
by *metis*

lemma *DotPure*:
assumes *pure A*
shows *Mult p A*, Δ ≡ *A*
by (*simp add*: *assms equivalent-def pure-mult1 pure-mult2*)

lemma *WildPure*:
assumes *pure A*
shows *Wildcard A*, Δ ≡ *A*
proof (*rule equivalentI*)
show ⋀σ *s*. σ, *s*, Δ ⊨ *Wildcard A* ⟹ σ, *s*, Δ ⊨ *A*
using *assms pure-def sat.simps*(*12*) **by** *blast*
show ⋀σ *s*. σ, *s*, Δ ⊨ *A* ⟹ σ, *s*, Δ ⊨ *Wildcard A*
by (*metis one-neutral sat.simps*(*12*))
qed

3.9 DotFull

lemma *mult-one-same1*:
Mult one A, $\Delta \vdash A$
by (*simp add*: *entails-def one-neutral*)

lemma *mult-one-same2*:
A, $\Delta \vdash$ *Mult one A*
by (*simp add*: *entailsI one-neutral*)

lemma *DotFull*:
Mult one A, $\Delta \equiv A$
using *equivalent-def mult-one-same1 mult-one-same2* **by** *blast*

3.10 DotExists

lemma *dot-exists1*:
Mult p (*Exists x A*), $\Delta \vdash$ *Exists x* (*Mult p A*)
proof (*rule entailsI*)
fix σ s Δ
assume σ, s, $\Delta \models$ *Mult p* (*Exists x A*)
then show σ, s, $\Delta \models$ *Exists x* (*Mult p A*)
by *auto*
qed

lemma *dot-exists2*:
Exists x (*Mult p A*), $\Delta \vdash$ *Mult p* (*Exists x A*)
proof (*rule entailsI*)
fix σ s Δ
assume σ, s, $\Delta \models$ *Exists x* (*Mult p A*)
then show σ, s, $\Delta \models$ *Mult p* (*Exists x A*) **by** *auto*
qed

lemma *DotExists*:
Mult p (*Exists x A*), $\Delta \equiv$ *Exists x* (*Mult p A*)
by (*simp add*: *dot-exists1 dot-exists2 equivalent-def*)

lemma *WildExists*:
Wildcard (*Exists x A*), $\Delta \equiv$ *Exists x* (*Wildcard A*)
proof (*rule equivalentI*)
show $\bigwedge\sigma$ *s*. σ, s, $\Delta \models$ *Wildcard* (*Exists x A*) $\Longrightarrow$ σ, s, $\Delta \models$ *Exists x* (*Wildcard A*)
by *auto*
show $\bigwedge\sigma$ *s*. σ, s, $\Delta \models$ *Exists x* (*Wildcard A*) $\Longrightarrow$ σ, s, $\Delta \models$ *Wildcard* (*Exists x A*)
by *auto*
qed

3.11 DotForall

lemma *dot-forall1*:
Mult p (*Forall x A*), Δ ⊢ *Forall x* (*Mult p A*)
proof (*rule entailsI*)
fix σ *s* Δ
assume σ, *s*, Δ ⊨ *Mult p* (*Forall x A*)
then show σ, *s*, Δ ⊨ *Forall x* (*Mult p A*)
by *auto*
qed

lemma *dot-forall2*:
Forall x (*Mult p A*), Δ ⊢ *Mult p* (*Forall x A*)
proof (*rule entailsI*)
fix σ *s* Δ
assume σ, *s*, Δ ⊨ *Forall x* (*Mult p A*)
obtain *a* **where** σ = *p* ⊙ *a*
using *sat.simps*(*1*) *sat-mult* **by** *blast*
have *a*, *s*, Δ ⊨ *Forall x A*
proof (*rule sat-forall*)
fix *v* **show** *a*, *s*(*x* := *v*), Δ ⊨ *A*
using ‹σ = *p* ⊙ *a*› ‹σ, *s*, Δ ⊨ *Forall x* (*Mult p A*)› *can-divide* **by** *auto*
qed
then show σ, *s*, Δ ⊨ *Mult p* (*Forall x A*)
using ‹σ = *p* ⊙ *a*› **by** *auto*
qed

lemma *DotForall*:
Mult p (*Forall x A*), Δ ≡ *Forall x* (*Mult p A*)
by (*simp add*: *dot-forall1 dot-forall2 equivalent-def*)

lemma *WildForall*:
Wildcard (*Forall x A*), Δ ⊢ *Forall x* (*Wildcard A*)
by (*metis* (*no-types*, *lifting*) *entailsI sat.simps*(*12*) *sat.simps*(*9*))

3.12 Split

lemma *split*:
Mult (*sadd a b*) *A*, Δ ⊢ *Star* (*Mult a A*) (*Mult b A*)
proof (*rule entailsI*)
fix σ *s*
assume σ, *s*, Δ ⊨ *Mult* (*sadd a b*) *A*
then show σ, *s*, Δ ⊨ *Star* (*Mult a A*) (*Mult b A*)
using *distrib-mult* **by** *fastforce*
qed

end

end

4 Combinability

This section corresponds to Section 3 of the paper [5].

theory *Combinability*
  imports *UnboundedLogic*
begin

context *logic*
begin

The definition of combinable assertions corresponds to Definition 4 of the paper [5].

definition *combinable* :: ('*d*, '*c*, '*a*) *interp* ⇒ ('*a*, '*b*, '*c*, '*d*) *assertion* ⇒ *bool*
where
  combinable Δ *A* ⟷ (∀ *p q*. *Star* (*Mult p A*) (*Mult q A*), Δ ⊢ *Mult* (*sadd p q*) *A*)

lemma *combinable-instantiate*:
  assumes *combinable* Δ *A*
    and *a*, *s*, Δ ⊨ *A*
    and *b*, *s*, Δ ⊨ *A*
    and *Some x* = *p* ⊙ *a* ⊕ *q* ⊙ *b*
   shows *x*, *s*, Δ ⊨ *Mult* (*sadd p q*) *A*
   by (*meson assms*(*1*) *assms*(*2*) *assms*(*3*) *assms*(*4*) *combinable-def entails-def logic.sat.simps*(*2*) *logic-axioms sat.simps*(*1*))

lemma *combinable-instantiate-one*:
  assumes *combinable* Δ *A*
    and *a*, *s*, Δ ⊨ *A*
    and *b*, *s*, Δ ⊨ *A*
    and *Some x* = *p* ⊙ *a* ⊕ *q* ⊙ *b*
    and *sadd p q* = *one*
   shows *x*, *s*, Δ ⊨ *A*
   using *assms*(*1*) *assms*(*2*) *assms*(*3*) *assms*(*4*) *assms*(*5*) *combinable-instantiate one-neutral* **by** *fastforce*

lemma *combinableI-old*:
  assumes ⋀*a b p q x σ s*. *a*, *s*, Δ ⊨ *A* ∧ *b*, *s*, Δ ⊨ *A* ∧ *Some σ* = *p* ⊙ *a* ⊕ *q* ⊙ *b* ∧ *σ* = (*sadd p q*) ⊙ *x* ⟹ *x*, *s*, Δ ⊨ *A*
  shows *combinable* Δ *A*
proof −
  have ⋀*p q*. *Star* (*Mult p A*) (*Mult q A*), Δ ⊢ *Mult* (*sadd p q*) *A*
  proof (*rule entailsI*)
    fix *p q σ s*
    assume *σ*, *s*, Δ ⊨ *Star* (*Mult p A*) (*Mult q A*)
    then obtain *a b* **where** *a*, *s*, Δ ⊨ *A* ∧ *b*, *s*, Δ ⊨ *A* ∧ *Some σ* = *p* ⊙ *a* ⊕ *q* ⊙ *b*
      by *auto*
    moreover obtain *x* **where** *σ* = (*sadd p q*) ⊙ *x*

using *unique-inv* **by** *auto*
ultimately have x, s, $\Delta \models A$ **using** *assms*
by *blast*
then show σ, s, $\Delta \models Mult\ (sadd\ p\ q)\ A$
using ‹$\sigma = sadd\ p\ q \odot x$› **by** *fastforce*
qed
then show *?thesis*
by (*simp add*: *combinable-def*)
qed

lemma *combinableI*:
assumes $\bigwedge a\ b\ p\ q\ x\ \sigma\ s.\ a, s, \Delta \models A \wedge b, s, \Delta \models A \wedge Some\ x = p \odot a \oplus q \odot b \wedge sadd\ p\ q = one \Longrightarrow x, s, \Delta \models A$
shows *combinable* $\Delta\ A$
proof (*rule combinableI-old*)
fix $a\ b\ p\ q\ x\ \sigma\ s$
assume $a, s, \Delta \models A \wedge b, s, \Delta \models A \wedge Some\ \sigma = p \odot a \oplus q \odot b \wedge \sigma = sadd\ p\ q \odot x$
let *?p* = $smult\ (sinv\ (sadd\ p\ q))\ p$
let *?q* = $smult\ (sinv\ (sadd\ p\ q))\ q$
have $Some\ x = ?p \odot a \oplus ?q \odot b$
proof −
have $Some\ ((smult\ (sinv\ (sadd\ p\ q))\ (sadd\ p\ q)) \odot x) = ?p \odot a \oplus ?q \odot b$
by (*metis* ‹$a, s, \Delta \models A \wedge b, s, \Delta \models A \wedge Some\ \sigma = p \odot a \oplus q \odot b \wedge \sigma = sadd\ p\ q \odot x$› *double-mult logic.plus-mult logic-axioms*)
then show *?thesis*
by (*simp add*: *one-neutral sinv-inverse smult-comm*)
qed
moreover have $sadd\ ?p\ ?q = one$
by (*metis logic.smult-comm logic-axioms sinv-inverse smult-distrib*)
ultimately show $x, s, \Delta \models A$
using ‹$a, s, \Delta \models A \wedge b, s, \Delta \models A \wedge Some\ \sigma = p \odot a \oplus q \odot b \wedge \sigma = sadd\ p\ q \odot x$› *assms* **by** *blast*
qed

lemma *combinable-wand*:
assumes *combinable* $\Delta\ B$
shows *combinable* $\Delta\ (Wand\ A\ B)$
proof (*rule combinableI-old*)
fix $a\ b\ p\ q\ x\ \sigma\ s$
assume $a, s, \Delta \models Wand\ A\ B \wedge b, s, \Delta \models Wand\ A\ B \wedge Some\ \sigma = p \odot a \oplus q \odot b \wedge \sigma = sadd\ p\ q \odot x$
show $x, s, \Delta \models Wand\ A\ B$
proof (*rule sat-wand*)
fix $aa\ \sigma'$
assume $aa, s, \Delta \models A \wedge Some\ \sigma' = x \oplus aa$
then have $Some\ ((sadd\ p\ q) \odot \sigma') = \sigma \oplus ((sadd\ p\ q) \odot aa)$

by (*simp add*: ‹*a*, *s*, Δ ⊨ *Wand A B* ∧ *b*, *s*, Δ ⊨ *Wand A B* ∧ *Some* σ = *p* ⊙ *a* ⊕ *q* ⊙ *b* ∧ σ = *sadd p q* ⊙ *x*› *plus-mult*)
moreover have *Some* ((*sadd p q*) ⊙ *aa*) = *p* ⊙ *aa* ⊕ *q* ⊙ *aa*
by (*simp add*: *distrib-mult*)
moreover have *a* ## *aa*
proof −
have *p* ⊙ *a* ## (*sadd p q*) ⊙ *aa*
by (*metis* ‹*a*, *s*, Δ ⊨ *Wand A B* ∧ *b*, *s*, Δ ⊨ *Wand A B* ∧ *Some* σ = *p* ⊙ *a* ⊕ *q* ⊙ *b* ∧ σ = *sadd p q* ⊙ *x*› *asso2 calculation*(1) *commutative compatible-def option.discI*)
then show *?thesis*
using *compatible-multiples* **by** *blast*
qed
then obtain *aaa* **where** *Some aaa* = *a* ⊕ *aa*
using *compatible-def* **by** *auto*
moreover have *b* ## *aa*
proof −
have *q* ⊙ *b* ## (*sadd p q*) ⊙ *aa*
by (*metis* ‹*a*, *s*, Δ ⊨ *Wand A B* ∧ *b*, *s*, Δ ⊨ *Wand A B* ∧ *Some* σ = *p* ⊙ *a* ⊕ *q* ⊙ *b* ∧ σ = *sadd p q* ⊙ *x*› *asso2 calculation*(1) *compatible-def option.discI*)
then show *?thesis*
using *compatible-multiples* **by** *blast*
qed
then obtain *baa* **where** *Some baa* = *b* ⊕ *aa*
using *compatible-def* **by** *auto*
ultimately have *Some* (*mult* (*sadd p q*) σ′) = *p* ⊙ *aaa* ⊕ *q* ⊙ *baa*
proof −
obtain *a1* **where** *Some a1* = σ ⊕ (*p* ⊙ *aa*)
by (*metis* ‹*Some* (*sadd p q* ⊙ σ′) = σ ⊕ *sadd p q* ⊙ *aa*› *compatible-multiples option.exhaust-sel pre-logic.compatible-def unique-inv*)
then obtain *a2* **where** *Some a2* = *p* ⊙ *a* ⊕ (*p* ⊙ *aa*)
by (*meson* ‹⋀*thesis*. (⋀*aaa*. *Some aaa* = *a* ⊕ *aa* ⟹ *thesis*) ⟹ *thesis*› *plus-mult*)
then have *Some a1* = *a2* ⊕ *q* ⊙ *b*
proof −
obtain *bc* **where** *q* ⊙ *b* ⊕ *p* ⊙ *aa* = *Some bc*
by (*metis* ‹*b* ## *aa*› *compatible-iff compatible-multiples one-neutral option.exhaust-sel pre-logic.compatible-def*)
then have σ ⊕ *p* ⊙ *aa* = *p* ⊙ *a* ⊕ *bc*
using *asso1*[*of p* ⊙ *a q* ⊙ *b* σ *p* ⊙ *aa bc*]
by (*metis* ‹*a*, *s*, Δ ⊨ *Wand A B* ∧ *b*, *s*, Δ ⊨ *Wand A B* ∧ *Some* σ = *p* ⊙ *a* ⊕ *q* ⊙ *b* ∧ σ = *sadd p q* ⊙ *x*›)
then show *?thesis*
by (*metis* ‹*Some a1* = σ ⊕ *p* ⊙ *aa*› ‹*Some a2* = *p* ⊙ *a* ⊕ *p* ⊙ *aa*› ‹*q* ⊙ *b* ⊕ *p* ⊙ *aa* = *Some bc*› *asso1 commutative*)
qed
moreover have *a2* = *p* ⊙ *aaa*
by (*metis* ‹*Some a2* = *p* ⊙ *a* ⊕ *p* ⊙ *aa*› ‹*Some aaa* = *a* ⊕ *aa*› *option.inject plus-mult*)

moreover have *Some* $(q \odot baa) = q \odot b \oplus q \odot aa$
by (*simp add*: ‹*Some baa* = $b \oplus aa$› *plus-mult*)
ultimately show *?thesis*
by (*metis* ‹*Some* (*sadd p q* $\odot$ σ') = $\sigma \oplus$ *sadd p q* $\odot$ *aa*› ‹*Some* (*sadd p q* $\odot$ *aa*) = $p \odot aa \oplus q \odot aa$› ‹*Some a1* = $\sigma \oplus p \odot aa$› *asso1*)
qed
moreover have *aaa*, *s*, $\Delta \models B \wedge$ *baa*, *s*, $\Delta \models B$
using ‹*Some aaa* = $a \oplus aa$› ‹*Some baa* = $b \oplus aa$› ‹*a*, *s*, $\Delta \models$ *Wand A B* $\wedge$ *b*, *s*, $\Delta \models$ *Wand A B* $\wedge$ *Some* $\sigma = p \odot a \oplus q \odot b \wedge \sigma =$ *sadd p q* $\odot$ *x*› ‹*aa*, *s*, $\Delta \models A \wedge$ *Some* $\sigma' = x \oplus aa$› **by** *auto*
ultimately have *mult* (*sadd p q*) σ', *s*, $\Delta \models$ *Mult* (*sadd p q*) *B*
by (*meson assms logic.combinable-def logic.entails-def logic-axioms sat.simps*(*1*) *sat.simps*(*2*))
then show σ', *s*, $\Delta \models B$
using *can-divide sat.simps*(*1*) **by** *metis*
qed
qed

lemma *combinable-star*:
assumes *combinable* Δ *A*
and *combinable* Δ *B*
shows *combinable* Δ (*Star A B*)
proof (*rule combinableI-old*)
fix *a b p q x* σ *s*
assume *a*, *s*, $\Delta \models$ *Star A B* $\wedge$ *b*, *s*, $\Delta \models$ *Star A B* $\wedge$ *Some* $\sigma = p \odot a \oplus q \odot b \wedge \sigma =$ *sadd p q* $\odot$ *x*
then obtain *aa ab ba bb* **where** *Some a* = *aa* $\oplus$ *ab Some b* = *ba* $\oplus$ *bb aa*, *s*, $\Delta \models A$
ab, *s*, $\Delta \models B$ *ba*, *s*, $\Delta \models A$ *bb*, *s*, $\Delta \models B$
by *auto*
then obtain *xa xb* **where** *Some xa* = $p \odot aa \oplus q \odot ba$ *Some xb* = $p \odot ab \oplus q \odot bb$
by (*metis* ‹*a*, *s*, $\Delta \models$ *Star A B* $\wedge$ *b*, *s*, $\Delta \models$ *Star A B* $\wedge$ *Some* $\sigma = p \odot a \oplus q \odot b \wedge \sigma =$ *sadd p q* $\odot$ *x*› *asso2 commutative compatible-iff compatible-multiples one-neutral option.discI option.exhaust-sel pre-logic.compatible-def*)
then have *xa*, *s*, $\Delta \models$ *Mult* (*sadd p q*) *A*
by (*meson* ‹*aa*, *s*, $\Delta \models A$› ‹*ba*, *s*, $\Delta \models A$› *assms*(*1*) *entails-def logic.combinable-def logic.sat.simps*(*1*) *logic.sat.simps*(*2*) *logic-axioms*)
moreover have *xb*, *s*, $\Delta \models$ *Mult* (*sadd p q*) *B*
by (*meson* ‹*Some xb* = $p \odot ab \oplus q \odot bb$› ‹*ab*, *s*, $\Delta \models B$› ‹*bb*, *s*, $\Delta \models B$› *assms*(*2*) *combinable-def entails-def sat.simps*(*1*) *sat.simps*(*2*))
moreover have *Some* $\sigma =$ *xa* $\oplus$ *xb*
using ‹*Some a* = *aa* $\oplus$ *ab*› ‹*Some b* = *ba* $\oplus$ *bb*› ‹*Some xa* = $p \odot aa \oplus q \odot ba$› ‹*Some xb* = $p \odot ab \oplus q \odot bb$› ‹*a*, *s*, $\Delta \models$ *Star A B* $\wedge$ *b*, *s*, $\Delta \models$ *Star A B* $\wedge$ *Some* $\sigma = p \odot a \oplus q \odot b \wedge \sigma =$ *sadd p q* $\odot$ *x*› *move-sum plus-mult* **by** *blast*
then obtain *xa' xb'* **where** *Some x* = *xa'* $\oplus$ *xb' xa* = *sadd p q* $\odot$ *xa' xb* = *sadd p q* $\odot$ *xb'*
by (*metis* ‹*a*, *s*, $\Delta \models$ *Star A B* $\wedge$ *b*, *s*, $\Delta \models$ *Star A B* $\wedge$ *Some* $\sigma = p \odot a \oplus q \odot b \wedge \sigma =$ *sadd p q* $\odot$ *x*› *plus-mult unique-inv*)

ultimately show x, s, $\Delta \models \mathit{Star}\ A\ B$
by (*metis logic.can-divide logic-axioms sat.simps*(*1*) *sat.simps*(*2*))
qed

lemma *combinable-mult*:
assumes *combinable* $\Delta\ A$
shows *combinable* Δ (*Mult* $\pi\ A$)
proof (*rule combinableI*)
fix $a\ b\ p\ q\ x\ \sigma\ s$
assume *asm*: a, s, $\Delta \models \mathit{Mult}\ \pi\ A \wedge b$, s, $\Delta \models \mathit{Mult}\ \pi\ A \wedge \mathit{Some}\ x = p \odot a \oplus q \odot b \wedge \mathit{sadd}\ p\ q = \mathit{one}$
then obtain $a'\ b'$ **where** a', s, $\Delta \models A\ b'$, s, $\Delta \models A\ a = \pi \odot a'\ b = \pi \odot b'$ **by** *auto*

let *?p* = *smult p* π
let *?q* = *smult q* π

have $\mathit{Some}\ x = ?p \odot a' \oplus ?q \odot b'$
by (*simp add*: ‹$a = \pi \odot a'$› ‹$b = \pi \odot b'$› *asm double-mult*)
moreover have *sadd ?p ?q* = π
using *asm smult-comm smult-distrib sone-neutral* **by** *force*
ultimately show x, s, $\Delta \models \mathit{Mult}\ \pi\ A$
by (*metis* ‹a', s, $\Delta \models A$› ‹b', s, $\Delta \models A$› *assms combinable-instantiate*)
qed

lemma *combinable-and*:
assumes *combinable* $\Delta\ A$
and *combinable* $\Delta\ B$
shows *combinable* Δ (*And A B*)
proof (*rule combinableI*)
fix $a\ b\ p\ q\ x\ \sigma\ s$
assume a, s, $\Delta \models \mathit{And}\ A\ B \wedge b$, s, $\Delta \models \mathit{And}\ A\ B \wedge \mathit{Some}\ x = p \odot a \oplus q \odot b \wedge \mathit{sadd}\ p\ q = \mathit{one}$
then obtain a, s, $\Delta \models A\ b$, s, $\Delta \models A\ a$, s, $\Delta \models B\ b$, s, $\Delta \models B$ **by** *auto*
then show x, s, $\Delta \models \mathit{And}\ A\ B$
by (*meson* ‹a, s, $\Delta \models \mathit{And}\ A\ B \wedge b$, s, $\Delta \models \mathit{And}\ A\ B \wedge \mathit{Some}\ x = p \odot a \oplus q \odot b \wedge \mathit{sadd}\ p\ q = \mathit{one}$› *assms*(*1*) *assms*(*2*) *combinable-instantiate-one sat.simps*(*7*))
qed

lemma *combinable-forall*:
assumes *combinable* $\Delta\ A$
shows *combinable* Δ (*Forall x A*)
proof (*rule combinableI*)
fix $a\ b\ p\ q\ y\ \sigma\ s$
assume a, s, $\Delta \models \mathit{Forall}\ x\ A \wedge b$, s, $\Delta \models \mathit{Forall}\ x\ A \wedge \mathit{Some}\ y = p \odot a \oplus q \odot b \wedge \mathit{sadd}\ p\ q = \mathit{one}$
show y, s, $\Delta \models \mathit{Forall}\ x\ A$
proof (*rule sat-forall*)

fix v **show** y, $s(x := v)$, $\Delta \models A$
by (*meson* ‹a, s, $\Delta \models$ *Forall* x $A \wedge b$, s, $\Delta \models$ *Forall* x $A \wedge$ *Some* $y = p \odot a \oplus q \odot b \wedge$ *sadd* p q = *one*› *assms combinable-instantiate-one sat.simps*(9))
qed
qed

definition *unambiguous* **where**
unambiguous Δ A $x \longleftrightarrow (\forall \sigma 1\ \sigma 2\ v1\ v2\ s.\ \sigma 1\ \#\#\ \sigma 2 \wedge \sigma 1, s(x := v1), \Delta \models A \wedge \sigma 2, s(x := v2), \Delta \models A \longrightarrow v1 = v2)$

lemma *unambiguousI*:
assumes $\bigwedge \sigma 1\ \sigma 2\ v1\ v2\ s.\ \sigma 1\ \#\#\ \sigma 2 \wedge \sigma 1, s(x := v1), \Delta \models A \wedge \sigma 2, s(x := v2), \Delta \models A \Longrightarrow v1 = v2$
shows *unambiguous* Δ A x
by (*simp add*: *assms unambiguous-def*)

lemma *unambiguous-star*:
assumes *unambiguous* Δ A x
shows *unambiguous* Δ (*Star* A B) x
proof (*rule unambiguousI*)
fix $\sigma 1\ \sigma 2\ v1\ v2\ s$
assume $\sigma 1\ \#\#\ \sigma 2 \wedge \sigma 1, s(x := v1), \Delta \models$ *Star* A $B \wedge \sigma 2, s(x := v2), \Delta \models$ *Star* A B
then obtain $a1\ b1\ a2\ b2$ **where** *Some* $\sigma 1 = a1 \oplus b1$ *Some* $\sigma 2 = a2 \oplus b2$ $a1, s(x := v1), \Delta \models A$
$a2, s(x := v2), \Delta \models A\ b1, s(x := v1), \Delta \models B\ b2, s(x := v2), \Delta \models B$ **by** *auto*
then have $a1\ \#\#\ a2$
by (*metis* ‹$\sigma 1\ \#\#\ \sigma 2 \wedge \sigma 1, s(x := v1), \Delta \models$ *Star* A $B \wedge \sigma 2, s$ $(x := v2), \Delta \models$ *Star* A B› *asso2 asso3 commutative*)
then show $v1 = v2$
using ‹$a1, s(x := v1), \Delta \models A$› ‹$a2, s(x := v2), \Delta \models A$› *assms unambiguous-def*
by *fastforce*
qed

lemma *combinable-exists*:
assumes *combinable* Δ A
and *unambiguous* Δ A x
shows *combinable* Δ (*Exists* x A)
proof (*rule combinableI*)
fix $a\ b\ p\ q\ y\ \sigma\ s$
assume $a, s, \Delta \models$ *Exists* x $A \wedge b, s, \Delta \models$ *Exists* x $A \wedge$ *Some* $y = p \odot a \oplus q \odot b \wedge$ *sadd* p q = *one*
then have $a\ \#\#\ b$
by (*metis logic.compatible-multiples logic-axioms option.discI pre-logic.compatible-def*)
moreover obtain $v1\ v2$ **where** $a, s(x := v1), \Delta \models A\ b, s(x := v2), \Delta \models A$

using ‹a, s, $\Delta \models$ *Exists* x $A \wedge$ b, s, $\Delta \models$ *Exists* x $A \wedge$ *Some* $y = p \odot a \oplus q \odot b \wedge$ *sadd* p q = *one*› **by** *auto*
ultimately have *v1* = *v2*
using *assms*(*2*) *unambiguous-def* **by** *force*
then show y, s, $\Delta \models$ *Exists* x A
by (*metis* (*mono-tags*, *opaque-lifting*) ‹a, s(x := *v1*), $\Delta \models A$› ‹a, s, $\Delta \models$ *Exists* x $A \wedge$ b, s, $\Delta \models$ *Exists* x $A \wedge$ *Some* $y = p \odot a \oplus q \odot b \wedge$ *sadd* p q = *one*› ‹b, s(x := *v2*), $\Delta \models A$› *assms*(*1*) *combinable-instantiate-one logic.sat.simps*(*8*) *logic-axioms*)
qed

lemma *combinable-pure*:
assumes *pure* A
shows *combinable* Δ A
using *assms combinableI-old pure-def* **by** *blast*

lemma *combinable-imp*:
assumes *pure* A
and *combinable* Δ B
shows *combinable* Δ (*Imp* A B)
proof (*rule combinableI*)
fix a b p q x σ s
assume a, s, $\Delta \models$ *Imp* A $B \wedge$ b, s, $\Delta \models$ *Imp* A $B \wedge$ *Some* $x = p \odot a \oplus q \odot b \wedge$ *sadd* p q = *one*
then show x, s, $\Delta \models$ *Imp* A B
using *assms*(*1*) *assms*(*2*) *combinable-instantiate-one pure-def sat.simps*(*5*)
by *metis*
qed

lemma *combinable-wildcard*:
assumes *combinable* Δ A
shows *combinable* Δ (*Wildcard* A)
proof (*rule combinableI*)
fix a b p q x σ s
assume *asm*: a, s, $\Delta \models$ *Wildcard* $A \wedge$ b, s, $\Delta \models$ *Wildcard* $A \wedge$ *Some* $x = p \odot a \oplus q \odot b \wedge$ *sadd* p q = *one*
then obtain a' b' *pa pb* **where** a', s, $\Delta \models A$ b', s, $\Delta \models A$ $a = pa \odot a'$ $b = pb \odot b'$ **by** *auto*
then have *Some* x = (*smult* p *pa*) $\odot$ $a' \oplus$ (*smult* q *pb*) $\odot$ b'
by (*simp add*: *asm double-mult*)
then have x, s, $\Delta \models$ *Mult* (*sadd* (*smult* p *pa*) (*smult* q *pb*)) A
using ‹a', s, $\Delta \models A$› ‹b', s, $\Delta \models A$› *assms combinable-instantiate* **by** *blast*
then show x, s, $\Delta \models$ *Wildcard* A
by *fastforce*
qed

end

end

5 (Co)Inductive Predicates

This subsection corresponds to Section 4 of the paper [5].

theory *FixedPoint*
imports *Distributivity Combinability*
begin

type-synonym $('d, 'c, 'a)$ *chain* $=$ *nat* $\Rightarrow$ $('d, 'c, 'a)$ *interp*

context *logic*
begin

5.1 Definitions

definition *smaller-interp* :: $('d, 'c, 'a)$ *interp* $\Rightarrow$ $('d, 'c, 'a)$ *interp* $\Rightarrow$ *bool* **where**
smaller-interp Δ Δ' $\longleftrightarrow$ $(\forall s.\ \Delta\ s \subseteq \Delta'\ s)$

lemma *smaller-interpI*:
assumes $\bigwedge s\ x.\ x \in \Delta\ s \Longrightarrow x \in \Delta'\ s$
shows *smaller-interp* Δ Δ'
by (*simp add*: *assms smaller-interp-def subsetI*)

definition *indep-interp* **where**
indep-interp $A \longleftrightarrow (\forall x\ s\ \Delta\ \Delta'.\ x,\ s,\ \Delta \models A \longleftrightarrow x,\ s,\ \Delta' \models A)$

fun *applies-eq* :: $('a, 'b, 'c, 'd)$ *assertion* $\Rightarrow$ $('d, 'c, 'a)$ *interp* $\Rightarrow$ $('d, 'c, 'a)$ *interp*
where
applies-eq A Δ $s = \{\ a\ |a.\ a,\ s,\ \Delta \models A\ \}$

definition *monotonic* :: $(('d, 'c, 'a)$ *interp* $\Rightarrow$ $('d, 'c, 'a)$ *interp*$) \Rightarrow$ *bool* **where**
monotonic $f \longleftrightarrow (\forall \Delta\ \Delta'.$ *smaller-interp* Δ $\Delta' \longrightarrow$ *smaller-interp* $(f\ \Delta)$ $(f\ \Delta'))$

lemma *monotonicI*:
assumes $\bigwedge \Delta\ \Delta'.$ *smaller-interp* Δ $\Delta' \Longrightarrow$ *smaller-interp* $(f\ \Delta)$ $(f\ \Delta')$
shows *monotonic* f
by (*simp add*: *assms monotonic-def*)

definition *non-increasing* :: $(('d, 'c, 'a)$ *interp* $\Rightarrow$ $('d, 'c, 'a)$ *interp*$) \Rightarrow$ *bool* **where**
non-increasing $f \longleftrightarrow (\forall \Delta\ \Delta'.$ *smaller-interp* Δ $\Delta' \longrightarrow$ *smaller-interp* $(f\ \Delta')$ $(f\ \Delta))$

lemma *non-increasingI*:
assumes $\bigwedge \Delta\ \Delta'.$ *smaller-interp* Δ $\Delta' \Longrightarrow$ *smaller-interp* $(f\ \Delta')$ $(f\ \Delta)$
shows *non-increasing* f

by (*simp add*: *assms non-increasing-def*)

lemma *smaller-interp-refl*:
smaller-interp Δ Δ
by (*simp add*: *smaller-interp-def*)

lemma *smaller-interp-applies-cons*:
assumes *smaller-interp* (*applies-eq A* Δ) (*applies-eq A* Δ′)
and *a*, *s*, Δ ⊨ *A*
shows *a*, *s*, Δ′ ⊨ *A*
proof −
have *a* ∈ *applies-eq A* Δ *s*
using *assms*(*2*) **by** *force*
then have *a* ∈ *applies-eq A* Δ′ *s*
by (*metis assms*(*1*) *in-mono smaller-interp-def*)
then show *?thesis* **by** *auto*
qed

definition *empty-interp* **where**
empty-interp s = {}

definition *full-interp* :: ('*d*, '*c*, '*a*) *interp* **where**
full-interp s = *UNIV*

lemma *smaller-interp-trans*:
assumes *smaller-interp a b*
and *smaller-interp b c*
shows *smaller-interp a c*
by (*metis assms*(*1*) *assms*(*2*) *dual-order.trans smaller-interp-def*)

lemma *smaller-empty*:
smaller-interp empty-interp x
by (*simp add*: *empty-interp-def smaller-interp-def*)

The definition of set-closure properties corresponds to Definition 8 of the paper [5].

definition *set-closure-property* :: ('*a* ⇒ '*a* ⇒ '*a set*) ⇒ ('*d*, '*c*, '*a*) *interp* ⇒ *bool* **where**
set-closure-property S Δ ⟷ (∀ *a b s*. *a* ∈ Δ *s* ∧ *b* ∈ Δ *s* ⟶ *S a b* ⊆ Δ *s*)

lemma *set-closure-propertyI*:
assumes ⋀*a b s*. *a* ∈ Δ *s* ∧ *b* ∈ Δ *s* ⟹ *S a b* ⊆ Δ *s*
shows *set-closure-property S* Δ
by (*simp add*: *assms set-closure-property-def*)

lemma *set-closure-property-instantiate*:
assumes *set-closure-property S* Δ

and $a \in \Delta\ s$
and $b \in \Delta\ s$
and $x \in S\ a\ b$
shows $x \in \Delta\ s$
using *assms subsetD set-closure-property-def* **by** *metis*

5.2 Everything preserves monotonicity

lemma *indep-implies-non-increasing*:
assumes *indep-interp A*
shows *non-increasing* (*applies-eq A*)
by (*metis* (*no-types*, *lifting*) *applies-eq.simps assms indep-interp-def smaller-interp-def mem-Collect-eq non-increasingI subsetI*)

5.2.1 Monotonicity

lemma *mono-instantiate*:
assumes *monotonic* (*applies-eq A*)
and $x \in$ *applies-eq A* $\Delta\ s$
and *smaller-interp* $\Delta\ \Delta'$
shows $x \in$ *applies-eq A* $\Delta'\ s$
using *assms*(*1*) *assms*(*2*) *assms*(*3*) *monotonic-def smaller-interp-applies-cons*
by *fastforce*

lemma *mono-star*:
assumes *monotonic* (*applies-eq A*)
and *monotonic* (*applies-eq B*)
shows *monotonic* (*applies-eq* (*Star A B*))
proof (*rule monotonicI*)
fix $\Delta\ \Delta' :: ('c, 'd, 'a)$ *interp*
assume *asm0*: *smaller-interp* $\Delta\ \Delta'$
show *smaller-interp* (*applies-eq* (*Star A B*) Δ) (*applies-eq* (*Star A B*) Δ')
proof (*rule smaller-interpI*)
fix *s x* **assume** *asm1*: $x \in$ *applies-eq* (*Star A B*) $\Delta\ s$
then obtain *a b* **where** *Some* $x = a \oplus b\ a \in$ *applies-eq A* $\Delta\ s\ b \in$ *applies-eq B* $\Delta\ s$
by *auto*
then have $a \in$ *applies-eq A* $\Delta'\ s \wedge b \in$ *applies-eq B* $\Delta'\ s$
by (*meson asm0 assms*(*1*) *assms*(*2*) *mono-instantiate*)
then show $x \in$ *applies-eq* (*Star A B*) $\Delta'\ s$
using ‹*Some* $x = a \oplus b$› **by** *force*
qed
qed

lemma *mono-wand*:
assumes *non-increasing* (*applies-eq A*)
and *monotonic* (*applies-eq B*)
shows *monotonic* (*applies-eq* (*Wand A B*))
proof (*rule monotonicI*)

```
  fix Δ Δ′ :: ('c, 'd, 'a) interp
  assume asm0: smaller-interp Δ Δ′
  show smaller-interp (applies-eq (Wand A B) Δ) (applies-eq (Wand A B) Δ′)
  proof (rule smaller-interpI)
    fix s x assume asm1: x ∈ applies-eq (Wand A B) Δ s
    have x, s, Δ′ ⊨ Wand A B
    proof (rule sat-wand)
      fix a b
      assume asm2: a, s, Δ′ ⊨ A ∧ Some b = x ⊕ a
      then have a, s, Δ ⊨ A
        by (meson asm0 assms(1) non-increasing-def smaller-interp-applies-cons)
      then have b, s, Δ ⊨ B
        using asm1 asm2 by auto
      then show b, s, Δ′ ⊨ B
        by (meson asm0 assms(2) monotonic-def smaller-interp-applies-cons)
    qed
    then show x ∈ applies-eq (Wand A B) Δ′ s
      by simp
  qed
qed



lemma mono-and:
  assumes monotonic (applies-eq A)
      and monotonic (applies-eq B)
    shows monotonic (applies-eq (And A B))
proof (rule monotonicI)
  fix Δ Δ′ :: ('c, 'd, 'a) interp
  assume asm0: smaller-interp Δ Δ′
  show smaller-interp (applies-eq (And A B) Δ) (applies-eq (And A B) Δ′)
  proof (rule smaller-interpI)
    fix s x assume asm1: x ∈ applies-eq (And A B) Δ s
    then show x ∈ applies-eq (And A B) Δ′ s
      using asm0 assms(1) assms(2) monotonic-def logic-axioms mem-Collect-eq
sat.simps(8) smaller-interp-applies-cons by fastforce
  qed
qed

lemma mono-or:
  assumes monotonic (applies-eq A)
      and monotonic (applies-eq B)
    shows monotonic (applies-eq (Or A B))
proof (rule monotonicI)
  fix Δ Δ′ :: ('c, 'd, 'a) interp
  assume asm0: smaller-interp Δ Δ′
  show smaller-interp (applies-eq (Or A B) Δ) (applies-eq (Or A B) Δ′)
  proof (rule smaller-interpI)
    fix s x assume asm1: x ∈ applies-eq (Or A B) Δ s
```

```
    then show x ∈ applies-eq (Or A B) Δ' s
      using asm0 assms(1) assms(2) monotonic-def logic-axioms mem-Collect-eq
sat.simps(8) smaller-interp-applies-cons by fastforce
  qed
qed

lemma mono-sem:
  monotonic (applies-eq (Sem B))
  using monotonic-def smaller-interp-def by fastforce

lemma mono-interp:
  monotonic (applies-eq Pred)
proof (rule monotonicI)
  fix Δ Δ' :: ('c, 'd, 'a) interp
  assume smaller-interp Δ Δ'
  show smaller-interp (applies-eq Pred Δ) (applies-eq Pred Δ')
  proof (rule smaller-interpI)
    fix s x assume x ∈ applies-eq Pred Δ s
    then show x ∈ applies-eq Pred Δ' s
     by (metis (mono-tags, lifting) ‹smaller-interp Δ Δ'› applies-eq.simps in-mono
mem-Collect-eq sat.simps(10) smaller-interp-def)
  qed
qed


lemma mono-mult:
  assumes monotonic (applies-eq A)
  shows monotonic (applies-eq (Mult π A))
proof (rule monotonicI)
  fix Δ Δ' :: ('c, 'd, 'a) interp
  assume asm0: smaller-interp Δ Δ'
  show smaller-interp (applies-eq (Mult π A) Δ) (applies-eq (Mult π A) Δ')
  proof (rule smaller-interpI)
    fix s x assume asm1: x ∈ applies-eq (Mult π A) Δ s
    then show x ∈ applies-eq (Mult π A) Δ' s
      using asm0 assms monotonic-def smaller-interp-applies-cons by fastforce
  qed
qed

lemma mono-wild:
  assumes monotonic (applies-eq A)
  shows monotonic (applies-eq (Wildcard A))
proof (rule monotonicI)
  fix Δ Δ' :: ('c, 'd, 'a) interp
  assume asm0: smaller-interp Δ Δ'
  show smaller-interp (applies-eq (Wildcard A) Δ) (applies-eq (Wildcard A) Δ')
  proof (rule smaller-interpI)
    fix s x assume asm1: x ∈ applies-eq (Wildcard A) Δ s
    then show x ∈ applies-eq (Wildcard A) Δ' s
```

```
    using asm0 assms monotonic-def smaller-interp-applies-cons by fastforce
  qed
qed


lemma mono-imp:
  assumes non-increasing (applies-eq A)
      and monotonic (applies-eq B)
    shows monotonic (applies-eq (Imp A B))
proof (rule monotonicI)
  fix Δ Δ' :: ('c, 'd, 'a) interp
  assume asm0: smaller-interp Δ Δ'
  show smaller-interp (applies-eq (Imp A B) Δ) (applies-eq (Imp A B) Δ')
  proof (rule smaller-interpI)
    fix s x assume asm1: x ∈ applies-eq (Imp A B) Δ s
    have x, s, Δ' ⊨ Imp A B
    proof (cases x, s, Δ' ⊨ A)
      case True
      then have x, s, Δ ⊨ A
        by (meson asm0 assms(1) non-increasing-def smaller-interp-applies-cons)
      then have x, s, Δ ⊨ B
        using asm1 by auto
      then show ?thesis
      by (metis asm0 assms(2) monotonic-def sat.simps(5) smaller-interp-applies-cons)
    next
      case False
      then show ?thesis by simp
    qed
    then show x ∈ applies-eq (Imp A B) Δ' s
      by simp
  qed
qed

lemma mono-bounded:
  assumes monotonic (applies-eq A)
  shows monotonic (applies-eq (Bounded A))
proof (rule monotonicI)
  fix Δ Δ' :: ('c, 'd, 'a) interp
  assume asm: smaller-interp Δ Δ'
  show smaller-interp (applies-eq (Bounded A) Δ) (applies-eq (Bounded A) Δ')
  proof (rule smaller-interpI)
    fix s x assume x ∈ applies-eq (Bounded A) Δ s
    then show x ∈ applies-eq (Bounded A) Δ' s
      using asm assms monotonic-def smaller-interp-applies-cons by fastforce
  qed
qed

lemma mono-exists:
  assumes monotonic (applies-eq A)
```

shows *monotonic* (*applies-eq* (*Exists v A*))
proof (*rule monotonicI*)
fix Δ Δ′ :: ('*c*, '*d*, '*a*) *interp*
assume *asm0*: *smaller-interp* Δ Δ′
show *smaller-interp* (*applies-eq* (*Exists v A*) Δ) (*applies-eq* (*Exists v A*) Δ′)
proof (*rule smaller-interpI*)
fix *s x* **assume** *asm1*: *x* ∈ *applies-eq* (*Exists v A*) Δ *s*
then show *x* ∈ *applies-eq* (*Exists v A*) Δ′ *s*
using *asm0 assms monotonic-def smaller-interp-applies-cons* **by** *fastforce*
qed
qed

lemma *mono-forall*:
assumes *monotonic* (*applies-eq A*)
shows *monotonic* (*applies-eq* (*Forall v A*))
proof (*rule monotonicI*)
fix Δ Δ′ :: ('*c*, '*d*, '*a*) *interp*
assume *asm0*: *smaller-interp* Δ Δ′
show *smaller-interp* (*applies-eq* (*Forall v A*) Δ) (*applies-eq* (*Forall v A*) Δ′)
proof (*rule smaller-interpI*)
fix *s x* **assume** *asm1*: *x* ∈ *applies-eq* (*Forall v A*) Δ *s*
then show *x* ∈ *applies-eq* (*Forall v A*) Δ′ *s*
using *asm0 assms monotonic-def smaller-interp-applies-cons* **by** *fastforce*
qed
qed

5.2.2 Non-increasing

lemma *non-increasing-instantiate*:
assumes *non-increasing* (*applies-eq A*)
and *x* ∈ *applies-eq A* Δ′ *s*
and *smaller-interp* Δ Δ′
shows *x* ∈ *applies-eq A* Δ *s*
using *assms*(*1*) *assms*(*2*) *assms*(*3*) *non-increasing-def smaller-interp-applies-cons*
by *fastforce*

lemma *non-inc-star*:
assumes *non-increasing* (*applies-eq A*)
and *non-increasing* (*applies-eq B*)
shows *non-increasing* (*applies-eq* (*Star A B*))
proof (*rule non-increasingI*)
fix Δ Δ′ :: ('*c*, '*d*, '*a*) *interp*
assume *asm0*: *smaller-interp* Δ Δ′
show *smaller-interp* (*applies-eq* (*Star A B*) Δ′) (*applies-eq* (*Star A B*) Δ)
proof (*rule smaller-interpI*)
fix *s x* **assume** *asm1*: *x* ∈ *applies-eq* (*Star A B*) Δ′ *s*
then obtain *a b* **where** *Some x* = *a* ⊕ *b a* ∈ *applies-eq A* Δ′ *s b* ∈ *applies-eq B* Δ′ *s*

```
    by auto
  then have a ∈ applies-eq A Δ s ∧ b ∈ applies-eq B Δ s
    by (meson asm0 assms(1) assms(2) non-increasing-instantiate)
  then show x ∈ applies-eq (Star A B) Δ s
    using ‹Some x = a ⊕ b› by force
 qed
qed


lemma non-increasing-wand:
 assumes monotonic (applies-eq A)
    and non-increasing (applies-eq B)
  shows non-increasing (applies-eq (Wand A B))
proof (rule non-increasingI)
 fix Δ Δ' :: ('c, 'd, 'a) interp
 assume asm0: smaller-interp Δ Δ'
 show smaller-interp (applies-eq (Wand A B) Δ') (applies-eq (Wand A B) Δ)
 proof (rule smaller-interpI)
  fix s x assume asm1: x ∈ applies-eq (Wand A B) Δ' s
  have x, s, Δ ⊨ Wand A B
  proof (rule sat-wand)
   fix a b
   assume asm2: a, s, Δ ⊨ A ∧ Some b = x ⊕ a
   then have a, s, Δ' ⊨ A
    by (meson asm0 assms(1) monotonic-def smaller-interp-applies-cons)
   then have b, s, Δ' ⊨ B
    using asm1 asm2 by auto
   then show b, s, Δ ⊨ B
    by (meson asm0 assms(2) non-increasing-def smaller-interp-applies-cons)
  qed
  then show x ∈ applies-eq (Wand A B) Δ s
   by simp
 qed
qed


lemma non-increasing-and:
 assumes non-increasing (applies-eq A)
    and non-increasing (applies-eq B)
  shows non-increasing (applies-eq (And A B))
proof (rule non-increasingI)
 fix Δ Δ' :: ('c, 'd, 'a) interp
 assume asm0: smaller-interp Δ' Δ
 show smaller-interp (applies-eq (And A B) Δ) (applies-eq (And A B) Δ')
 proof (rule smaller-interpI)
  fix s x assume asm1: x ∈ applies-eq (And A B) Δ s
  then show x ∈ applies-eq (And A B) Δ' s
   using asm0 assms(1) assms(2) non-increasing-def logic-axioms mem-Collect-eq
```

sat.simps(*8*) *smaller-interp-applies-cons* **by** *fastforce*
qed
qed

lemma *non-increasing-or*:
assumes *non-increasing* (*applies-eq A*)
and *non-increasing* (*applies-eq B*)
shows *non-increasing* (*applies-eq* (*Or A B*))
proof (*rule non-increasingI*)
fix Δ Δ′ :: (′*c*, ′*d*, ′*a*) *interp*
assume *asm0*: *smaller-interp* Δ Δ′
show *smaller-interp* (*applies-eq* (*Or A B*) Δ′) (*applies-eq* (*Or A B*) Δ)
proof (*rule smaller-interpI*)
fix *s x* **assume** *asm1*: *x* ∈ *applies-eq* (*Or A B*) Δ′ *s*
then show *x* ∈ *applies-eq* (*Or A B*) Δ *s*
using *asm0 assms*(*1*) *assms*(*2*) *non-increasing-def logic-axioms mem-Collect-eq*
sat.simps(*8*) *smaller-interp-applies-cons* **by** *fastforce*
qed
qed

lemma *non-increasing-sem*:
non-increasing (*applies-eq* (*Sem B*))
using *non-increasing-def smaller-interp-def* **by** *fastforce*

lemma *non-increasing-mult*:
assumes *non-increasing* (*applies-eq A*)
shows *non-increasing* (*applies-eq* (*Mult* π *A*))
proof (*rule non-increasingI*)
fix Δ Δ′ :: (′*c*, ′*d*, ′*a*) *interp*
assume *asm0*: *smaller-interp* Δ Δ′
show *smaller-interp* (*applies-eq* (*Mult* π *A*) Δ′) (*applies-eq* (*Mult* π *A*) Δ)
proof (*rule smaller-interpI*)
fix *s x* **assume** *asm1*: *x* ∈ *applies-eq* (*Mult* π *A*) Δ′ *s*
then show *x* ∈ *applies-eq* (*Mult* π *A*) Δ *s*
using *asm0 assms non-increasing-def smaller-interp-applies-cons* **by** *fastforce*
qed
qed

lemma *non-increasing-wild*:
assumes *non-increasing* (*applies-eq A*)
shows *non-increasing* (*applies-eq* (*Wildcard A*))
proof (*rule non-increasingI*)
fix Δ Δ′ :: (′*c*, ′*d*, ′*a*) *interp*
assume *asm0*: *smaller-interp* Δ Δ′
show *smaller-interp* (*applies-eq* (*Wildcard A*) Δ′) (*applies-eq* (*Wildcard A*) Δ)
proof (*rule smaller-interpI*)
fix *s x* **assume** *asm1*: *x* ∈ *applies-eq* (*Wildcard A*) Δ′ *s*
then show *x* ∈ *applies-eq* (*Wildcard A*) Δ *s*

```
    using asm0 assms non-increasing-def smaller-interp-applies-cons by fastforce
  qed
qed


lemma non-increasing-imp:
  assumes monotonic (applies-eq A)
      and non-increasing (applies-eq B)
    shows non-increasing (applies-eq (Imp A B))
proof (rule non-increasingI)
  fix Δ Δ' :: ('c, 'd, 'a) interp
  assume asm0: smaller-interp Δ Δ'
  show smaller-interp (applies-eq (Imp A B) Δ') (applies-eq (Imp A B) Δ)
  proof (rule smaller-interpI)
    fix s x assume asm1: x ∈ applies-eq (Imp A B) Δ' s
    have x, s, Δ ⊨ Imp A B
    proof (cases x, s, Δ ⊨ A)
      case True
      then have x, s, Δ' ⊨ A
        by (meson asm0 assms(1) monotonic-def smaller-interp-applies-cons)
      then have x, s, Δ' ⊨ B
        using asm1 by auto
      then show ?thesis
      by (metis asm0 assms(2) non-increasing-def sat.simps(5) smaller-interp-applies-cons)
    next
      case False
      then show ?thesis by simp
    qed
    then show x ∈ applies-eq (Imp A B) Δ s
      by simp
  qed
qed


lemma non-increasing-bounded:
  assumes non-increasing (applies-eq A)
  shows non-increasing (applies-eq (Bounded A))
proof (rule non-increasingI)
  fix Δ Δ' :: ('c, 'd, 'a) interp
  assume asm: smaller-interp Δ' Δ
  show smaller-interp (applies-eq (Bounded A) Δ) (applies-eq (Bounded A) Δ')
  proof (rule smaller-interpI)
    fix s x assume x ∈ applies-eq (Bounded A) Δ s
    then show x ∈ applies-eq (Bounded A) Δ' s
      using asm assms non-increasing-def smaller-interp-applies-cons by fastforce
  qed
qed
```

lemma *non-increasing-exists*:
assumes *non-increasing* (*applies-eq A*)
shows *non-increasing* (*applies-eq* (*Exists v A*))
proof (*rule non-increasingI*)
fix Δ Δ′ :: (′*c*, ′*d*, ′*a*) *interp*
assume *asm0*: *smaller-interp* Δ′ Δ
show *smaller-interp* (*applies-eq* (*Exists v A*) Δ) (*applies-eq* (*Exists v A*) Δ′)
proof (*rule smaller-interpI*)
fix *s x* **assume** *asm1*: *x* ∈ *applies-eq* (*Exists v A*) Δ *s*
then show *x* ∈ *applies-eq* (*Exists v A*) Δ′ *s*
using *asm0 assms non-increasing-def smaller-interp-applies-cons* **by** *fastforce*
qed
qed

lemma *non-increasing-forall*:
assumes *non-increasing* (*applies-eq A*)
shows *non-increasing* (*applies-eq* (*Forall v A*))
proof (*rule non-increasingI*)
fix Δ Δ′ :: (′*c*, ′*d*, ′*a*) *interp*
assume *asm0*: *smaller-interp* Δ′ Δ
show *smaller-interp* (*applies-eq* (*Forall v A*) Δ) (*applies-eq* (*Forall v A*) Δ′)
proof (*rule smaller-interpI*)
fix *s x* **assume** *asm1*: *x* ∈ *applies-eq* (*Forall v A*) Δ *s*
then show *x* ∈ *applies-eq* (*Forall v A*) Δ′ *s*
using *asm0 assms non-increasing-def smaller-interp-applies-cons* **by** *fastforce*
qed
qed

5.3 Tarski's fixed points

5.3.1 Greatest Fixed Point

definition *D* :: ((′*d*, ′*c*, ′*a*) *interp* ⇒ (′*d*, ′*c*, ′*a*) *interp*) ⇒ (′*d*, ′*c*, ′*a*) *interp set*
where
D f = { Δ |Δ. *smaller-interp* Δ (*f* Δ) }

fun *GFP* :: ((′*d*, ′*c*, ′*a*) *interp* ⇒ (′*d*, ′*c*, ′*a*) *interp*) ⇒ (′*d*, ′*c*, ′*a*) *interp* **where**
GFP f s = { σ |σ. ∃Δ ∈ *D f*. σ ∈ Δ *s* }

lemma *smaller-interp-D*:
assumes *x* ∈ *D f*
shows *smaller-interp x* (*GFP f*)
by (*metis* (*mono-tags*, *lifting*) *CollectI GFP.elims assms smaller-interpI*)

lemma *GFP-lub*:
assumes ⋀*x*. *x* ∈ *D f* ⟹ *smaller-interp x y*
shows *smaller-interp* (*GFP f*) *y*
proof (*rule smaller-interpI*)
fix *s x*

assume $x \in GFP\ f\ s$
then obtain Δ **where** $\Delta \in D\ f\ x \in \Delta\ s$
by *auto*
then show $x \in y\ s$
by (*metis assms in-mono smaller-interp-def*)
qed

lemma *smaller-interp-antisym*:
assumes *smaller-interp a b*
and *smaller-interp b a*
shows $a = b$
proof (*rule ext*)
fix x **show** $a\ x = b\ x$
by (*metis assms(1) assms(2) set-eq-subset smaller-interp-def*)
qed

5.3.2 Least Fixed Point

definition *DD* :: $(('d, 'c, 'a)\ interp \Rightarrow ('d, 'c, 'a)\ interp) \Rightarrow ('d, 'c, 'a)\ interp\ set$
where
$DD\ f = \{\ \Delta\ |\Delta.\ smaller\text{-}interp\ (f\ \Delta)\ \Delta\ \}$

fun *LFP* :: $(('d, 'c, 'a)\ interp \Rightarrow ('d, 'c, 'a)\ interp) \Rightarrow ('d, 'c, 'a)\ interp$ **where**
$LFP\ f\ s = \{\ \sigma\ |\sigma.\ \forall \Delta \in DD\ f.\ \sigma \in \Delta\ s\ \}$

lemma *smaller-interp-DD*:
assumes $x \in DD\ f$
shows *smaller-interp* $(LFP\ f)\ x$
using *assms smaller-interp-def* **by** *fastforce*

lemma *LFP-glb*:
assumes $\bigwedge x.\ x \in DD\ f \Longrightarrow smaller\text{-}interp\ y\ x$
shows *smaller-interp* $y\ (LFP\ f)$
proof (*rule smaller-interpI*)
fix $s\ x$
assume $x \in y\ s$
then have $\bigwedge \Delta.\ \Delta \in DD\ f \Longrightarrow x \in \Delta\ s$
by (*metis assms smaller-interp-def subsetD*)
then show $x \in LFP\ f\ s$
by *simp*
qed

5.4 Combinability and (an assertion being) intuitionistic are set-closure properties

5.4.1 Intuitionistic assertions

definition *sem-intui* :: $('d, 'c, 'a)\ interp \Rightarrow bool$ **where**
$sem\text{-}intui\ \Delta \longleftrightarrow (\forall s\ \sigma\ \sigma'.\ \sigma' \succeq \sigma \wedge \sigma \in \Delta\ s \longrightarrow \sigma' \in \Delta\ s)$

lemma *sem-intuiI*:
assumes $\bigwedge s\ \sigma\ \sigma'.\ \sigma' \succeq \sigma \wedge \sigma \in \Delta\ s \Longrightarrow \sigma' \in \Delta\ s$
shows *sem-intui* Δ
using *assms sem-intui-def* **by** *blast*

lemma *instantiate-intui-applies*:
assumes *intuitionistic s* Δ *A*
and $\sigma' \succeq \sigma$
and $\sigma \in$ *applies-eq A* Δ *s*
shows $\sigma' \in$ *applies-eq A* Δ *s*
using *assms*(*1*) *assms*(*2*) *assms*(*3*) *intuitionistic-def* **by** *fastforce*

lemma *sem-intui-intuitionistic*:
sem-intui (*applies-eq A* Δ) $\longleftrightarrow$ ($\forall$ *s. intuitionistic s* Δ *A*) (**is** *?A* $\longleftrightarrow$ *?B*)
proof
show *?B* $\Longrightarrow$ *?A*
proof −
assume *?B*
show *?A*
proof (*rule sem-intuiI*)
fix *s* σ σ'
assume $\sigma' \succeq \sigma \wedge \sigma \in$ *applies-eq A* Δ *s*
then show $\sigma' \in$ *applies-eq A* Δ *s*
using ‹$\forall$ *s. intuitionistic s* Δ *A*› *instantiate-intui-applies* **by** *blast*
qed
qed
assume *?A*
show *?B*
proof
fix *s* **show** *intuitionistic s* Δ *A*
proof (*rule intuitionisticI*)
fix *a b*
assume $a \succeq b \wedge b,\ s,\ \Delta \models A$
then have $b \in$ *applies-eq A* Δ *s* **by** *simp*
then show $a,\ s,\ \Delta \models A$
by (*metis CollectD* ‹$a \succeq b \wedge b,\ s,\ \Delta \models A$› ‹*sem-intui* (*applies-eq A* Δ)›
applies-eq.simps sem-intui-def)
qed
qed
qed

lemma *intuitionistic-set-closure*:
sem-intui = *set-closure-property* ($\lambda a\ b.\ \{\ \sigma\ |\sigma.\ \sigma \succeq a\}$)
proof (*rule ext*)
fix Δ :: (*'c*, *'d*, *'a*) *interp*
show *sem-intui* Δ = *set-closure-property* ($\lambda a\ b.\ \{\sigma\ |\sigma.\ \sigma \succeq a\}$) Δ (**is** *?A* $\longleftrightarrow$

?B)
proof
show *?A* $\Longrightarrow$ *?B*
by (*metis* (*no-types*, *lifting*) *CollectD set-closure-propertyI sem-intui-def subsetI*)
assume *?B*
show *?A*
proof (*rule sem-intuiI*)
fix s σ σ'
assume $\sigma' \succeq \sigma \wedge \sigma \in \Delta\ s$
moreover have $(\lambda a\ b.\ \{\sigma\ |\sigma.\ \sigma \succeq a\})\ \sigma\ \sigma = \{\sigma'\ |\sigma'.\ \sigma' \succeq \sigma\}$ **by** *simp*
ultimately have $\{\sigma'\ |\sigma'.\ \sigma' \succeq \sigma\} \subseteq \Delta\ s$
by (*metis* ‹*set-closure-property* $(\lambda a\ b.\ \{\sigma\ |\sigma.\ \sigma \succeq a\})\ \Delta$› *set-closure-property-def*)
show $\sigma' \in \Delta\ s$
using ‹$\sigma' \succeq \sigma \wedge \sigma \in \Delta\ s$› ‹$\{\sigma'\ |\sigma'.\ \sigma' \succeq \sigma\} \subseteq \Delta\ s$› **by** *fastforce*
qed
qed
qed

5.4.2 Combinable assertions

definition *sem-combinable* :: ('*d*, '*c*, '*a*) *interp* $\Rightarrow$ *bool* **where**
sem-combinable $\Delta \longleftrightarrow (\forall s\ p\ q\ a\ b\ x.\ sadd\ p\ q = one \wedge a \in \Delta\ s \wedge b \in \Delta\ s \wedge Some\ x = p \odot a \oplus q \odot b \longrightarrow x \in \Delta\ s)$

lemma *sem-combinableI*:
assumes $\bigwedge s\ p\ q\ a\ b\ x.\ sadd\ p\ q = one \wedge a \in \Delta\ s \wedge b \in \Delta\ s \wedge Some\ x = p \odot a \oplus q \odot b \Longrightarrow x \in \Delta\ s$
shows *sem-combinable* Δ
using *assms sem-combinable-def* **by** *blast*

lemma *sem-combinableE*:
assumes *sem-combinable* Δ
and $a \in \Delta\ s$
and $b \in \Delta\ s$
and $Some\ x = p \odot a \oplus q \odot b$
and $sadd\ p\ q = one$
shows $x \in \Delta\ s$
using *assms*(*1*) *assms*(*2*) *assms*(*3*) *assms*(*4*) *assms*(*5*) *sem-combinable-def*[*of* Δ]
by *blast*

lemma *applies-eq-equiv*:
$x \in applies\text{-}eq\ A\ \Delta\ s \longleftrightarrow x,\ s,\ \Delta \models A$
by *simp*

lemma *sem-combinable-appliesE*:
assumes *sem-combinable* (*applies-eq* $A\ \Delta$)
and $a,\ s,\ \Delta \models A$

and b, s, $\Delta \models A$
and *Some* $x = p \odot a \oplus q \odot b$
and *sadd* p q = *one*
shows x, s, $\Delta \models A$
using *sem-combinableE*[*of applies-eq* A Δ a s b x p q] *assms* **by** *simp*

lemma *sem-combinable-equiv*:
sem-combinable (*applies-eq* A Δ) $\longleftrightarrow$ (*combinable* Δ A) (**is** *?A* $\longleftrightarrow$ *?B*)
proof
show *?B* $\Longrightarrow$ *?A*
proof −
assume *?B*
show *?A*
proof (*rule sem-combinableI*)
fix s p q a b x
assume *asm*: *sadd* p q = *one* $\wedge$ $a \in$ *applies-eq* A Δ s $\wedge$ $b \in$ *applies-eq* A Δ s $\wedge$ *Some* $x = p \odot a \oplus q \odot b$
then show $x \in$ *applies-eq* A Δ s
using ‹*combinable* Δ A› *applies-eq-equiv combinable-instantiate-one* **by** *blast*
qed
qed
assume *?A*
show *?B*
proof −
fix s **show** *combinable* Δ A
proof (*rule combinableI*)
fix a b p q x σ s
assume a, s, $\Delta \models A \wedge b$, s, $\Delta \models A \wedge$ *Some* $x = p \odot a \oplus q \odot b \wedge$ *sadd* p q = *one*
then show x, s, $\Delta \models A$
using ‹*sem-combinable* (*applies-eq* A Δ)› *sem-combinable-appliesE* **by** *blast*
qed
qed
qed

lemma *combinable-set-closure*:
sem-combinable = *set-closure-property* ($\lambda a\ b$. { σ |σ p q. *sadd* p q = *one* $\wedge$ *Some* $\sigma = p \odot a \oplus q \odot b$})
proof (*rule ext*)
fix Δ :: ('*c*, '*d*, '*a*) *interp*
show *sem-combinable* Δ = *set-closure-property* ($\lambda a\ b$. { σ |σ p q. *sadd* p q = *one* $\wedge$ *Some* $\sigma = p \odot a \oplus q \odot b$}) Δ (**is** *?A* $\longleftrightarrow$ *?B*)
proof
show *?A* $\Longrightarrow$ *?B*
proof −
assume *?A*
show *?B*
proof (*rule set-closure-propertyI*)

fix a b s
assume $a \in \Delta\ s \wedge b \in \Delta\ s$
then show $\{x.\ \exists \sigma\ p\ q.\ x = \sigma \wedge sadd\ p\ q = one \wedge Some\ \sigma = p \odot a \oplus q \odot b\} \subseteq \Delta\ s$
using ‹*sem-combinable* Δ› *sem-combinableE* **by** *blast*
qed
qed
assume *?B*
show *?A*
proof (*rule sem-combinableI*)
fix s p q a b x
assume *asm*: $sadd\ p\ q = one \wedge a \in \Delta\ s \wedge b \in \Delta\ s \wedge Some\ x = p \odot a \oplus q \odot b$

then have $x \in (\lambda a\ b.\ \{\ \sigma\ |\sigma\ p\ q.\ sadd\ p\ q = one \wedge Some\ \sigma = p \odot a \oplus q \odot b\})\ a\ b$
by *blast*
moreover have $(\lambda a\ b.\ \{\ \sigma\ |\sigma\ p\ q.\ sadd\ p\ q = one \wedge Some\ \sigma = p \odot a \oplus q \odot b\})\ a\ b \subseteq \Delta\ s$
using ‹*?B*› *set-closure-property-def*[*of* $(\lambda a\ b.\ \{\ \sigma\ |\sigma\ p\ q.\ sadd\ p\ q = one \wedge Some\ \sigma = p \odot a \oplus q \odot b\})\ \Delta$]
asm **by** *meson*
ultimately show $x \in \Delta\ s$ **by** *blast*
qed
qed
qed

5.5 Transfinite induction

definition *Inf* :: $('d, 'c, 'a)\ interp\ set \Rightarrow ('d, 'c, 'a)\ interp$ **where**
$Inf\ S\ s = \{\ \sigma\ |\sigma.\ \forall \Delta \in S.\ \sigma \in \Delta\ s\}$

definition *Sup* :: $('d, 'c, 'a)\ interp\ set \Rightarrow ('d, 'c, 'a)\ interp$ **where**
$Sup\ S\ s = \{\ \sigma\ |\sigma.\ \exists \Delta \in S.\ \sigma \in \Delta\ s\}$

definition *inf* :: $('d, 'c, 'a)\ interp \Rightarrow ('d, 'c, 'a)\ interp \Rightarrow ('d, 'c, 'a)\ interp$ **where**
$inf\ \Delta\ \Delta'\ s = \Delta\ s \cap \Delta'\ s$

definition *less* **where**
$less\ a\ b \longleftrightarrow smaller\text{-}interp\ a\ b \wedge a \neq b$

definition *sup* :: $('d, 'c, 'a)\ interp \Rightarrow ('d, 'c, 'a)\ interp \Rightarrow ('d, 'c, 'a)\ interp$ **where**
$sup\ \Delta\ \Delta'\ s = \Delta\ s \cup \Delta'\ s$

lemma *smaller-full*:
smaller-interp x full-interp
by (*simp add*: *full-interp-def smaller-interpI*)

lemma *inf-empty*:
local.Inf {} = *full-interp*
proof (*rule ext*)
fix *s* :: *'c* ⇒ *'d* **show** *local.Inf* {} *s* = *full-interp s*
by (*simp add*: *Inf-def full-interp-def*)
qed

lemma *sup-empty*:
local.Sup {} = *empty-interp*
proof (*rule ext*)
fix *s* :: *'c* ⇒ *'d* **show** *local.Sup* {} *s* = *empty-interp s*
by (*simp add*: *Sup-def empty-interp-def*)
qed

lemma *test-axiom-inf*:
assumes ⋀*x*. *x* ∈ *A* ⟹ *smaller-interp z x*
shows *smaller-interp z* (*local.Inf A*)
proof (*rule smaller-interpI*)
fix *s x*
assume *x* ∈ *z s*
then have ⋀*y*. *y* ∈ *A* ⟹ *x* ∈ *y s*
by (*metis assms in-mono smaller-interp-def*)
then show *x* ∈ *local.Inf A s*
by (*simp add*: *Inf-def*)
qed

lemma *test-axiom-sup*:
assumes ⋀*x*. *x* ∈ *A* ⟹ *smaller-interp x z*
shows *smaller-interp* (*local.Sup A*) *z*
proof (*rule smaller-interpI*)
fix *s x*
assume *x* ∈ *local.Sup A s*
then obtain *y* **where** *y* ∈ *A x* ∈ *y s*
using *Sup-def*[*of A s*] *mem-Collect-eq*[*of x*]
by *auto*
then show *x* ∈ *z s*
by (*metis assms smaller-interp-def subsetD*)
qed

interpretation *complete-lattice Inf Sup inf smaller-interp less sup empty-interp full-interp*
apply *standard*
apply (*metis less-def smaller-interp-antisym*)
apply (*simp add*: *smaller-interp-refl*)
using *smaller-interp-trans* **apply** *blast*
using *smaller-interp-antisym* **apply** *blast*
apply (*simp add*: *inf-def smaller-interp-def*)

apply (*simp add*: *inf-def smaller-interp-def*)
apply (*simp add*: *inf-def smaller-interp-def*)
apply (*simp add*: *smaller-interpI sup-def*)
apply (*simp add*: *smaller-interpI sup-def*)
apply (*simp add*: *smaller-interp-def sup-def*)
apply (*metis* (*mono-tags*, *lifting*) *CollectD Inf-def smaller-interpI*)
using *test-axiom-inf* **apply** *blast*
apply (*metis* (*mono-tags*, *lifting*) *CollectI Sup-def smaller-interpI*)
using *test-axiom-sup* **apply** *auto*[*1*]
apply (*simp add*: *inf-empty*)
by (*simp add*: *sup-empty*)

lemma *mono-same*:
monotonic f $\longleftrightarrow$ *order-class.mono f*
by (*metis* (*no-types*, *opaque-lifting*) *le-funE le-funI monotonic-def order-class.mono-def smaller-interp-def*)

lemma *smaller-interp a b* $\longleftrightarrow$ $a \leq b$
by (*simp add*: *le-fun-def smaller-interp-def*)

lemma *set-closure-property-admissible*:
ccpo.admissible Sup-class.Sup ($\leq$) (*set-closure-property S*)
proof (*rule ccpo.admissibleI*)
fix *A* :: ('*c*, '*d*, '*a*) *interp set*
assume *asm0*: *Complete-Partial-Order.chain* ($\leq$) *A*
$A \neq \{\}$ $\forall x{\in}A.$ *set-closure-property S x*

show *set-closure-property S* (*Sup-class.Sup A*)
proof (*rule set-closure-propertyI*)
fix *a b s*
assume *asm*: $a \in$ *Sup-class.Sup A s* $\wedge$ $b \in$ *Sup-class.Sup A s*
then obtain Δa Δb **where** $\Delta a \in A$ $\Delta b \in A$ $a \in \Delta a\ s$ $b \in \Delta b\ s$
by *auto*
then show *S a b* $\subseteq$ *Sup-class.Sup A s*
proof (*cases* $\Delta a\ s \subseteq \Delta b\ s$)
case *True*
then have *S a b* $\subseteq \Delta b\ s$
by (*metis* ‹$\Delta b \in A$› ‹$a \in \Delta a\ s$› ‹$b \in \Delta b\ s$› *asm0*(*3*) *set-closure-property-def subsetD*)
then show *?thesis*
using ‹$\Delta b \in A$› **by** *auto*
next
case *False*
then have $\Delta b\ s \subseteq \Delta a\ s$
by (*metis* ‹$\Delta a \in A$› ‹$\Delta b \in A$› *asm0*(*1*) *chainD le-funD*)
then have *S a b* $\subseteq \Delta a\ s$
by (*metis* ‹$\Delta a \in A$› ‹$a \in \Delta a\ s$› ‹$b \in \Delta b\ s$› *asm0*(*3*) *subsetD set-closure-property-def*)

```
    then show ?thesis using ‹Δa ∈ A› by auto
   qed
  qed
qed


definition supp :: ('d, 'c, 'a) interp ⇒ bool where
  supp Δ ⟷ (∀ a b s. a ∈ Δ s ∧ b ∈ Δ s ⟶ (∃ x. a ≽ x ∧ b ≽ x ∧ x ∈ Δ s))

lemma suppI:
  assumes ⋀a b s. a ∈ Δ s ∧ b ∈ Δ s ⟹ (∃ x. a ≽ x ∧ b ≽ x ∧ x ∈ Δ s)
  shows supp Δ
  by (simp add: assms supp-def)

lemma supp-admissible:
  ccpo.admissible Sup-class.Sup (≤) supp
proof (rule ccpo.admissibleI)
  fix A :: ('c, 'd, 'a) interp set
  assume asm0: Complete-Partial-Order.chain (≤) A
  A ≠ {} ∀ x∈A. supp x
  show supp (Sup-class.Sup A)
  proof (rule suppI)
    fix a b s
    assume asm: a ∈ Sup-class.Sup A s ∧ b ∈ Sup-class.Sup A s
    then obtain Δa Δb where Δa ∈ A Δb ∈ A a ∈ Δa s b ∈ Δb s
      by auto
    then show ∃ x. a ≽ x ∧ b ≽ x ∧ x ∈ Sup-class.Sup A s
    proof (cases Δa s ⊆ Δb s)
      case True
      then have a ∈ Δb s
        using ‹a ∈ Δa s› by blast
      then obtain x where a ≽ x b ≽ x x ∈ Δb s
        by (metis ‹Δb ∈ A› ‹b ∈ Δb s› asm0(3) supp-def)
      then show ?thesis
        using ‹Δb ∈ A› by auto
    next
      case False
      then have b ∈ Δa s
       by (metis ‹Δa ∈ A› ‹Δb ∈ A› ‹b ∈ Δb s› asm0(1) chainD le-funD subsetD)
      then obtain x where a ≽ x b ≽ x x ∈ Δa s
        using ‹Δa ∈ A› ‹a ∈ Δa s› asm0(3) supp-def by metis
      then show ?thesis using ‹Δa ∈ A› by auto
    qed
  qed
qed

lemma Sup-class.Sup {} = empty-interp using empty-interp-def
  by fastforce
```

lemma *set-closure-prop-empty-all*:
shows *set-closure-property S empty-interp*
and *set-closure-property S full-interp*
apply (*metis empty-interp-def equals0D set-closure-propertyI*)
by (*simp add*: *full-interp-def set-closure-propertyI*)

lemma *LFP-preserves-set-closure-property-aux*:
assumes *monotonic f*
and *set-closure-property S empty-interp*
and $\bigwedge\Delta$. *set-closure-property S* $\Delta \Longrightarrow$ *set-closure-property S* (*f* Δ)
shows *set-closure-property S* (*ccpo-class.fixp f*)
using *set-closure-property-admissible*
proof (*rule fixp-induct*[*of set-closure-property S*])
show *set-closure-property S* (*Sup-class.Sup* {})
by (*simp add*: *set-closure-property-def*)
show *monotone* ($\leq$) ($\leq$) *f*
by (*metis* (*full-types*) *assms*(*1*) *le-fun-def monotoneI monotonic-def smaller-interp-def*)
show $\bigwedge x$. *set-closure-property S x* $\Longrightarrow$ *set-closure-property S* (*f x*)
by (*simp add*: *assms*(*3*))
qed

lemma *GFP-preserves-set-closure-property-aux*:
assumes *order-class.mono f*
and *set-closure-property S full-interp*
and $\bigwedge\Delta$. *set-closure-property S* $\Delta \Longrightarrow$ *set-closure-property S* (*f* Δ)
shows *set-closure-property S* (*complete-lattice-class.gfp f*)
using *assms*(*1*)
proof (*rule gfp-ordinal-induct*[*of f set-closure-property S*])
show $\bigwedge Sa$. *set-closure-property S Sa* $\Longrightarrow$ *complete-lattice-class.gfp f* $\leq$ *Sa* $\Longrightarrow$ *set-closure-property S* (*f Sa*)
using *assms*(*3*) **by** *blast*
fix *M* :: ('*c*, '*d*, '*a*) *interp set*
assume $\forall$ *Sa*$\in$*M*. *set-closure-property S Sa*
show *set-closure-property S* (*Inf-class.Inf M*)
proof (*rule set-closure-propertyI*)
fix *a b s*
assume *a* $\in$ *Inf-class.Inf M s* $\wedge$ *b* $\in$ *Inf-class.Inf M s*
then have $\bigwedge\Delta$. $\Delta \in M \Longrightarrow a \in \Delta\ s \wedge b \in \Delta\ s$
by *simp*
then have $\bigwedge\Delta$. $\Delta \in M \Longrightarrow S\ a\ b \subseteq \Delta\ s$
by (*metis* ‹$\forall$ *Sa*$\in$*M*. *set-closure-property S Sa*› *set-closure-property-def*)
show *S a b* $\subseteq$ *Inf-class.Inf M s*
by (*simp add*: ‹$\bigwedge\Delta$. $\Delta \in M \Longrightarrow S\ a\ b \subseteq \Delta\ s$› *complete-lattice-class.INF-greatest*)
qed
qed

5.6 Theorems

5.6.1 Greatest Fixed Point

theorem *GFP-is-FP*:
assumes *monotonic f*
shows *f* (*GFP f*) = *GFP f*
proof −
let *?u* = *GFP f*
have ⋀*x*. *x* ∈ *D f* ⟹ *smaller-interp x* (*f ?u*)
proof −
fix *x*
assume *x* ∈ *D f*
then have *smaller-interp* (*f x*) (*f ?u*)
using *assms monotonic-def smaller-interp-D* **by** *blast*
moreover have *smaller-interp x* (*f x*)
using *D-def* ‹*x* ∈ *D f*› **by** *fastforce*
ultimately show *smaller-interp x* (*f ?u*)
using *smaller-interp-trans* **by** *blast*
qed
then have *?u* ∈ *D f*
using *D-def GFP-lub* **by** *blast*
then have *f ?u* ∈ *D f*
by (*metis CollectI D-def* ‹⋀*x*. *x* ∈ *D f* ⟹ *smaller-interp x* (*f* (*GFP f*))› *assms monotonic-def*)
then show *?thesis*
by (*simp add*: ‹*GFP f* ∈ *D f*› ‹⋀*x*. *x* ∈ *D f* ⟹ *smaller-interp x* (*f* (*GFP f*))› *smaller-interp-D smaller-interp-antisym*)
qed

theorem *GFP-greatest*:
assumes *f u* = *u*
shows *smaller-interp u* (*GFP f*)
by (*simp add*: *D-def assms smaller-interp-D smaller-interp-refl*)

lemma *same-GFP*:
assumes *monotonic f*
shows *complete-lattice-class.gfp f* = *GFP f*
proof −
have *f* (*GFP f*) = *GFP f*
using *GFP-is-FP assms* **by** *blast*
then have *smaller-interp* (*GFP f*) (*complete-lattice-class.gfp f*)
by (*metis complete-lattice-class.gfp-upperbound le-funD order-class.order.eq-iff smaller-interp-def*)
moreover have *f* (*complete-lattice-class.gfp f*) = *complete-lattice-class.gfp f*
using *assms gfp-fixpoint mono-same* **by** *blast*
then have *smaller-interp* (*complete-lattice-class.gfp f*) (*GFP f*)
by (*simp add*: *GFP-greatest*)

ultimately show *?thesis*
by *simp*
qed

5.6.2 Least Fixed Point

theorem *LFP-is-FP*:
assumes *monotonic f*
shows *f* (*LFP f*) = *LFP f*
proof −
let *?u* = *LFP f*
have ⋀*x*. *x* ∈ *DD f* ⟹ *smaller-interp* (*f ?u*) *x*
proof −
fix *x*
assume *x* ∈ *DD f*
then have *smaller-interp* (*f ?u*) (*f x*)
using *assms monotonic-def smaller-interp-DD* **by** *blast*
moreover have *smaller-interp* (*f x*) *x*
using *DD-def* ‹*x* ∈ *DD f*› **by** *fastforce*
ultimately show *smaller-interp* (*f ?u*) *x*
using *smaller-interp-trans* **by** *blast*
qed
then have *?u* ∈ *DD f*
using *DD-def LFP-glb* **by** *blast*
then have *f ?u* ∈ *DD f*
by (*metis* (*mono-tags*, *lifting*) *CollectI DD-def* ‹⋀*x*. *x* ∈ *DD f* ⟹ *smaller-interp* (*f* (*LFP f*)) *x*› *assms monotonic-def*)
then show *?thesis*
by (*simp add*: ‹*LFP f* ∈ *DD f*› ‹⋀*x*. *x* ∈ *DD f* ⟹ *smaller-interp* (*f* (*LFP f*)) *x*› *smaller-interp-DD smaller-interp-antisym*)
qed

theorem *LFP-least*:
assumes *f u* = *u*
shows *smaller-interp* (*LFP f*) *u*
by (*simp add*: *DD-def assms smaller-interp-DD smaller-interp-refl*)

lemma *same-LFP*:
assumes *monotonic f*
shows *complete-lattice-class.lfp f* = *LFP f*
proof −
have *f* (*LFP f*) = *LFP f*
using *LFP-is-FP assms* **by** *blast*
then have *smaller-interp* (*complete-lattice-class.lfp f*) (*LFP f*)
by (*metis complete-lattice-class.lfp-lowerbound le-funE preorder-class.order-refl smaller-interp-def*)
moreover have *f* (*complete-lattice-class.gfp f*) = *complete-lattice-class.gfp f*

using *assms gfp-fixpoint mono-same* **by** *blast*
then have *smaller-interp* (*LFP f*) (*complete-lattice-class.lfp f*)
by (*meson LFP-least assms lfp-fixpoint mono-same*)
ultimately show *?thesis*
by *simp*
qed

lemma *LFP-same*:
assumes *monotonic f*
shows *ccpo-class.fixp f* = *LFP f*
proof −
have *f* (*ccpo-class.fixp f*) = *ccpo-class.fixp f*
by (*metis* (*mono-tags*, *lifting*) *assms fixp-unfold mono-same monotoneI order-class.mono-def*)
then have *smaller-interp* (*LFP f*) (*ccpo-class.fixp f*)
by (*simp add*: *LFP-least*)
moreover have *f* (*LFP f*) = *LFP f*
using *LFP-is-FP assms* **by** *blast*
then have *ccpo-class.fixp f* ≤ *LFP f*
by (*metis assms fixp-lowerbound mono-same monotoneI order-class.mono-def preorder-class.order-refl*)
ultimately show *?thesis*
by (*metis assms lfp-eq-fixp mono-same same-LFP*)
qed

The following theorem corresponds to Theorem 5 of the paper [5].

theorem *FP-preserves-set-closure-property*:
assumes *monotonic f*
and ⋀Δ. *set-closure-property S* Δ ⟹ *set-closure-property S* (*f* Δ)
shows *set-closure-property S* (*GFP f*)
and *set-closure-property S* (*LFP f*)
apply (*metis GFP-preserves-set-closure-property-aux assms*(*1*) *assms*(*2*) *mono-same same-GFP set-closure-prop-empty-all*(*2*))
by (*metis LFP-preserves-set-closure-property-aux LFP-same assms*(*1*) *assms*(*2*) *set-closure-prop-empty-all*(*1*))

end

end

6 Properties of Magic Wands

theory *WandProperties*
imports *Distributivity*
begin

context *logic*
begin

lemma *modus-ponens*:
Star P (*Wand P Q*), Δ ⊢ *Q*
proof (*rule entailsI*)
fix σ *s*
assume σ, *s*, Δ ⊨ *Star P* (*Wand P Q*)
show σ, *s*, Δ ⊨ *Q*
using ‹σ, *s*, Δ ⊨ *Star P* (*Wand P Q*)› *commutative* **by** *force*
qed

lemma *transitivity*:
Star (*Wand A B*) (*Wand B C*), Δ ⊢ *Wand A C*
proof (*rule entailsI*)
fix σ *s*
assume *asm0*: σ, *s*, Δ ⊨ *Star* (*Wand A B*) (*Wand B C*)
then obtain *ab bc* **where** *Some* σ = *ab* ⊕ *bc ab*, *s*, Δ ⊨ *Wand A B bc*, *s*, Δ ⊨ *Wand B C*
by *auto*
show σ, *s*, Δ ⊨ *Wand A C*
proof (*rule sat-wand*)
fix *a* σ′
assume *asm1*: *a*, *s*, Δ ⊨ *A* ∧ *Some* σ′ = σ ⊕ *a*
then obtain *aab* **where** *Some aab* = *ab* ⊕ *a*
by (*metis ‹Some* σ = *ab* ⊕ *bc›* *asso3 commutative compatible-def option.exhaust-sel*)
then have *Some* σ′ = *aab* ⊕ *bc*
by (*metis ‹Some* σ = *ab* ⊕ *bc›* *asm1 asso1 commutative*)
moreover have *aab*, *s*, Δ ⊨ *B*
using ‹*Some aab* = *ab* ⊕ *a*› ‹*ab*, *s*, Δ ⊨ *Wand A B*› *asm1* **by** *auto*
ultimately show σ′, *s*, Δ ⊨ *C*
using ‹*bc*, *s*, Δ ⊨ *Wand B C*› *commutative* **by** *auto*
qed
qed

lemma *currying1*:
Wand (*Star A B*) *C*, Δ ⊢ *Wand A* (*Wand B C*)
proof (*rule entailsI*)
fix σ *s*
assume *asm0*: σ, *s*, Δ ⊨ *Wand* (*Star A B*) *C*
show σ, *s*, Δ ⊨ *Wand A* (*Wand B C*)
proof (*rule sat-wand*)
fix *a* σ′
assume *asm1*: *a*, *s*, Δ ⊨ *A* ∧ *Some* σ′ = σ ⊕ *a*
show σ′, *s*, Δ ⊨ *Wand B C*
proof (*rule sat-wand*)
fix *b* σ″
assume *asm2*: *b*, *s*, Δ ⊨ *B* ∧ *Some* σ″ = σ′ ⊕ *b*
then obtain *ab* **where** *Some ab* = *a* ⊕ *b*
by (*metis asm1 asso2 compatible-def option.collapse*)
then have *ab*, *s*, Δ ⊨ *Star A B*

```
      using asm1 asm2 by auto
    moreover have Some σ'' = σ ⊕ ab
      by (metis ‹Some ab = a ⊕ b› asm1 asm2 asso1)
    ultimately show σ'', s, Δ ⊨ C
      using asm0 sat.simps(3) by blast
  qed
 qed
qed

lemma currying2:
  Wand A (Wand B C), Δ ⊢ Wand (Star A B) C
proof (rule entailsI)
 fix σ s
 assume asm0: σ, s, Δ ⊨ Wand A (Wand B C)
 show σ, s, Δ ⊨ Wand (Star A B) C
 proof (rule sat-wand)
  fix ab σ'
  assume asm1: ab, s, Δ ⊨ Star A B ∧ Some σ' = σ ⊕ ab
  then obtain a b where Some ab = a ⊕ b a, s, Δ ⊨ A b, s, Δ ⊨ B
    by auto
  then obtain bc where Some bc = σ ⊕ a
    by (metis asm1 asso3 compatible-def option.exhaust-sel)
  then have bc, s, Δ ⊨ Wand B C
    using ‹a, s, Δ ⊨ A› asm0 by auto
  moreover have Some σ' = bc ⊕ b
    by (metis ‹Some ab = a ⊕ b› ‹Some bc = σ ⊕ a› asm1 asso1)
  ultimately show σ', s, Δ ⊨ C
    using ‹b, s, Δ ⊨ B› sat.simps(3) by blast
 qed
qed

lemma distribution:
  Star (Wand A B) C, Δ ⊢ Wand A (Star B C)
proof (rule entailsI)
 fix σ s
 assume asm0: σ, s, Δ ⊨ Star (Wand A B) C
 then obtain ab c where Some σ = ab ⊕ c ab, s, Δ ⊨ Wand A B c, s, Δ ⊨ C
   by auto
 show σ, s, Δ ⊨ Wand A (Star B C)
 proof (rule sat-wand)
  fix a σ'
  assume asm1: a, s, Δ ⊨ A ∧ Some σ' = σ ⊕ a
  then obtain b where Some b = ab ⊕ a
  by (metis ‹Some σ = ab ⊕ c› asso3 commutative compatible-def option.exhaust-sel)
  then have b, s, Δ ⊨ B
    using ‹ab, s, Δ ⊨ Wand A B› asm1 by force
  moreover have Some σ' = b ⊕ c
    by (metis ‹Some σ = ab ⊕ c› ‹Some b = ab ⊕ a› asm1 asso1 commutative)
  ultimately show σ', s, Δ ⊨ Star B C
```

```
    using ‹c, s, Δ ⊨ C› sat.simps(2) by blast
  qed
qed

lemma adjunct1:
  assumes A, Δ ⊢ Wand B C
  shows Star A B, Δ ⊢ C
proof (rule entailsI)
  fix σ s
  assume σ, s, Δ ⊨ Star A B
  then show σ, s, Δ ⊨ C
    using assms entails-def by force
qed

lemma adjunct2:
  assumes Star A B, Δ ⊢ C
  shows A, Δ ⊢ Wand B C
proof (rule entailsI)
  fix σ s
  assume σ, s, Δ ⊨ A
  then show σ, s, Δ ⊨ Wand B C
    by (meson assms entails-def sat.simps(2) sat-wand)
qed


end

end
```

7 Fractional Predicates and Magic Wands in Automatic Separation Logic Verifiers

This section corresponds to Section 5 of the paper [5].

```
theory AutomaticVerifiers
  imports FixedPoint WandProperties
begin

context logic
begin
```

7.1 Syntactic multiplication

The following definition corresponds to Figure 6 of the paper [5].

```
fun syn-mult :: 'b ⇒ ('a, 'b, 'c, 'd) assertion ⇒ ('a, 'b, 'c, 'd) assertion where
  syn-mult π (Star A B) = Star (syn-mult π A) (syn-mult π B)
| syn-mult π (Wand A B) = Wand (syn-mult π A) (syn-mult π B)
| syn-mult π (Or A B) = Or (syn-mult π A) (syn-mult π B)
```

```
| syn-mult π (And A B) = And (syn-mult π A) (syn-mult π B)
| syn-mult π (Imp A B) = Imp (syn-mult π A) (syn-mult π B)
| syn-mult π (Mult α A) = syn-mult (smult α π) A
| syn-mult π (Exists x A) = Exists x (syn-mult π A)
| syn-mult π (Forall x A) = Forall x (syn-mult π A)
| syn-mult π (Wildcard A) = Wildcard A
| syn-mult π A = Mult π A

definition div-state where
  div-state π σ = (SOME r. σ = π ⊙ r)

lemma div-state-ok:
  σ = π ⊙ (div-state π σ)
  by (metis (mono-tags) div-state-def someI-ex unique-inv)
```

The following theorem corresponds to Theorem 6 of the paper [5].

```
theorem syn-sen-mult-same:
  σ, s, Δ ⊨ syn-mult π A ⟷ σ, s, Δ ⊨ Mult π A
proof (induct A arbitrary: σ π s)
  case (Exists x A)
  show ?case (is ?A ⟷ ?B)
  proof
    show ?B ⟹ ?A
      using Exists.hyps by auto
    show ?A ⟹ ?B
      using Exists.hyps by fastforce
  qed
next
  case (Forall x A)
  then show ?case
    by (metis dot-forall1 dot-forall2 entails-def sat.simps(9) syn-mult.simps(8))
next
  case (Star A B)
  show ?case (is ?P ⟷ ?Q)
  proof
    show ?P ⟹ ?Q
    proof −
      assume ?P
      then obtain a b where a, s, Δ ⊨ syn-mult π A b, s, Δ ⊨ syn-mult π B
      Some σ = a ⊕ b by auto
      then obtain a, s, Δ ⊨ Mult π A b, s, Δ ⊨ Mult π B
        using Star.hyps(1) Star.hyps(2) Star.prems by blast
      then show ?Q
        by (meson ‹Some σ = a ⊕ b› dot-star2 entails-def sat.simps(2))
    qed
    assume ?Q
    then obtain a b where a, s, Δ ⊨ Mult π A b, s, Δ ⊨ Mult π B Some σ = a
⊕ b
      by (meson dot-star1 entails-def sat.simps(2))
```

```
    then show ?P
      using Star.hyps(1) Star.hyps(2) Star.prems by force
  qed
next
  case (Mult p A)
  show ?case (is ?P ⟷ ?Q)
  proof
    show ?P ⟹ ?Q
    proof −
      assume ?P
      then have σ, s, Δ ⊨ syn-mult (smult p π) A by auto
      then have σ, s, Δ ⊨ Mult (smult p π) A
        using Mult.hyps by blast
      then show ?Q
        by (metis dot-mult2 logic.entails-def logic-axioms smult-comm)
    qed
    assume ?Q
    then obtain a where a, s, Δ ⊨ A σ = π ⊙ (p ⊙ a) by auto
    then show ?P
      using Mult.hyps double-mult smult-comm by auto
  qed
next
  case (Wand A B)
  show ?case (is ?P ⟷ ?Q)
  proof
    show ?P ⟹ ?Q
    proof −
      assume σ, s, Δ ⊨ syn-mult π (Wand A B)
      then have σ, s, Δ ⊨ Wand (syn-mult π A) (syn-mult π B)
        by auto
      moreover have div-state π σ, s, Δ ⊨ Wand A B
      proof (rule sat-wand)
        fix a b
        assume a, s, Δ ⊨ A ∧ Some b = div-state π σ ⊕ a
        then have Some (π ⊙ b) = σ ⊕ (π ⊙ a)
          using div-state-ok plus-mult by presburger
        moreover have π ⊙ a, s, Δ ⊨ Mult π A
          using ‹a, s, Δ ⊨ A ∧ Some b = div-state π σ ⊕ a› by auto
        then have π ⊙ a, s, Δ ⊨ syn-mult π A
          using Wand.hyps(1) Wand.prems by blast
        then have π ⊙ b, s, Δ ⊨ syn-mult π B
          using ‹σ, s, Δ ⊨ Wand (syn-mult π A) (syn-mult π B)› calculation by
auto
        ultimately show b, s, Δ ⊨ B
          by (metis Wand.hyps(2) Wand.prems can-divide sat.simps(1))
      qed
      then show σ, s, Δ ⊨ Mult π (Wand A B)
        by (metis div-state-ok sat.simps(1))
    qed
```

```
    assume σ, s, Δ ⊨ Mult π (Wand A B)
    then have div-state π σ, s, Δ ⊨ Wand A B
      by (metis div-state-ok can-divide sat.simps(1))
    have σ, s, Δ ⊨ Wand (syn-mult π A) (syn-mult π B)
    proof (rule sat-wand)
      fix a b assume a, s, Δ ⊨ syn-mult π A ∧ Some b = σ ⊕ a
      then have Some (div-state π b) = div-state π σ ⊕ div-state π a
        by (metis div-state-ok plus-mult unique-inv)
      then have div-state π b, s, Δ ⊨ B
          by (metis (no-types, lifting) Wand.hyps(1) ‹a, s, Δ ⊨ syn-mult π A ∧
Some b = σ ⊕ a› ‹div-state π σ, s, Δ ⊨ Wand A B› div-state-ok logic.can-divide
logic-axioms sat.simps(1) sat.simps(3))
      then show b, s, Δ ⊨ syn-mult π B
        using Wand.hyps(2) div-state-ok sat.simps(1) by blast
    qed
    then show σ, s, Δ ⊨ syn-mult π (Wand A B)
      by simp
  qed
next
  case (And A B)
  show ?case (is ?P ⟷ ?Q)
  proof
    show ?P ⟹ ?Q
    proof −
      assume ?P
      then obtain σ, s, Δ ⊨ syn-mult π A σ, s, Δ ⊨ syn-mult π B
        by auto
      then show ?Q
        by (meson And.hyps(1) And.hyps(2) dot-and2 logic.entails-def logic-axioms
sat.simps(7))
    qed
    assume ?Q then show ?P
      using And.hyps(1) And.hyps(2) And.prems by auto
  qed
next
  case (Imp A B)
  show ?case (is ?P ⟷ ?Q)
  proof
    show ?P ⟹ ?Q
    by (metis Imp.hyps(1) Imp.hyps(2) sat.simps(1) sat.simps(5) syn-mult.simps(5)
unique-inv)
    assume ?Q then show ?P
    by (metis Imp.hyps(1) Imp.hyps(2) Imp.prems can-divide sat.simps(1) sat.simps(5)
syn-mult.simps(5))
  qed
next
  case (Wildcard A)
  then show ?case
    by (metis DotWild entails-def equivalent-def syn-mult.simps(9))
```

qed (*auto*)

7.2 Monotonicity and fixed point

fun *pos-neg-rec-call* :: *bool* ⇒ ('*a*, '*b*, '*c*, '*d*) *assertion* ⇒ *bool* **where**
pos-neg-rec-call b Pred ⟷ *b*
| *pos-neg-rec-call b* (*Mult* - *A*) ⟷ *pos-neg-rec-call b A*
| *pos-neg-rec-call b* (*Exists* - *A*) ⟷ *pos-neg-rec-call b A*
| *pos-neg-rec-call b* (*Forall* - *A*) ⟷ *pos-neg-rec-call b A*
| *pos-neg-rec-call b* (*Star A B*) ⟷ *pos-neg-rec-call b A* ∧ *pos-neg-rec-call b B*
| *pos-neg-rec-call b* (*Or A B*) ⟷ *pos-neg-rec-call b A* ∧ *pos-neg-rec-call b B*
| *pos-neg-rec-call b* (*And A B*) ⟷ *pos-neg-rec-call b A* ∧ *pos-neg-rec-call b B*
| *pos-neg-rec-call b* (*Wand A B*) ⟷ *pos-neg-rec-call* (¬ *b*) *A* ∧ *pos-neg-rec-call b B*
| *pos-neg-rec-call b* (*Imp A B*) ⟷ *pos-neg-rec-call* (¬ *b*) *A* ∧ *pos-neg-rec-call b B*
| *pos-neg-rec-call* - (*Sem* -) ⟷ *True*
| *pos-neg-rec-call b* (*Bounded A*) ⟷ *pos-neg-rec-call b A*
| *pos-neg-rec-call b* (*Wildcard A*) ⟷ *pos-neg-rec-call b A*

lemma *pos-neg-rec-call-mono*:
assumes *pos-neg-rec-call b A*
shows (*b* ⟶ *monotonic* (*applies-eq A*)) ∧ (¬ *b* ⟶ *non-increasing* (*applies-eq A*))
using *assms*
proof (*induct A arbitrary*: *b*)
case (*Exists x A*)
then show *?case*
by (*meson mono-exists non-increasing-exists pos-neg-rec-call.simps*(*3*))
next
case (*Forall x A*)
then show *?case*
by (*meson mono-forall non-increasing-forall pos-neg-rec-call.simps*(*4*))
next
case (*Sem x*)
then show *?case*
by (*metis applies-eq.simps mem-Collect-eq mono-sem non-increasingI sat.simps*(*4*) *smaller-interp-def subsetI*)
next
case (*Mult x1a A*)
then show *?case*
using *mono-mult non-increasing-mult pos-neg-rec-call.simps*(*2*) **by** *blast*
next
case (*Star A1 A2*)
then show *?case*
by (*metis mono-star non-inc-star pos-neg-rec-call.simps*(*5*))
next
case (*Wand A1 A2*)
then show *?case*

by (*metis mono-wand non-increasing-wand pos-neg-rec-call.simps*(*8*))
next
case (*Or A1 A2*)
then show *?case*
by (*metis mono-or non-increasing-or pos-neg-rec-call.simps*(*6*))
next
case (*And A1 A2*)
then show *?case*
by (*metis mono-and non-increasing-and pos-neg-rec-call.simps*(*7*))
next
case (*Imp A1 A2*)
then show *?case*
by (*metis mono-imp non-increasing-imp pos-neg-rec-call.simps*(*9*))
next
case *Pred*
then show *?case*
using *mono-interp pos-neg-rec-call.simps*(*1*) **by** *blast*
next
case (*Bounded A*)
then show *?case*
using *mono-bounded non-increasing-bounded pos-neg-rec-call.simps*(*11*) **by** *blast*
next
case (*Wildcard A*)
then show *?case*
using *mono-wild non-increasing-wild pos-neg-rec-call.simps*(*12*) **by** *blast*
qed

The following theorem corresponds to Theorem 7 of the paper [5].

theorem *exists-lfp-gfp*:
assumes *pos-neg-rec-call True A*
shows σ, *s*, *LFP* (*applies-eq A*) $\models$ *A* $\longleftrightarrow$ $\sigma \in$ *LFP* (*applies-eq A*) *s*
and σ, *s*, *GFP* (*applies-eq A*) $\models$ *A* $\longleftrightarrow$ $\sigma \in$ *GFP* (*applies-eq A*) *s*
apply (*metis LFP-is-FP applies-eq.simps assms mem-Collect-eq pos-neg-rec-call-mono*)
by (*metis GFP-is-FP applies-eq.simps assms mem-Collect-eq pos-neg-rec-call-mono*)

7.3 Combinability

definition *combinable-sem* :: (($'d \Rightarrow 'c$) $\Rightarrow$ $'a \Rightarrow$ *bool*) $\Rightarrow$ *bool* **where**
combinable-sem B $\longleftrightarrow$ ($\forall$ *a b x s* α β. *B s a* $\wedge$ *B s b* $\wedge$ *sadd* α β = *one* $\wedge$ *Some x* = $\alpha \odot a \oplus \beta \odot b$ $\longrightarrow$ *B s x*)

fun *wf-assertion* :: ($'a$, $'b$, $'c$, $'d$) *assertion* $\Rightarrow$ *bool* **where**
wf-assertion Pred $\longleftrightarrow$ *True*
| *wf-assertion* (*Sem B*) $\longleftrightarrow$ *combinable-sem B*
| *wf-assertion* (*Mult - A*) $\longleftrightarrow$ *wf-assertion A*
| *wf-assertion* (*Forall - A*) $\longleftrightarrow$ *wf-assertion A*
| *wf-assertion* (*Exists x A*) $\longleftrightarrow$ *wf-assertion A* $\wedge$ ($\forall \Delta$. *unambiguous* Δ *A x*)
| *wf-assertion* (*Star A B*) $\longleftrightarrow$ *wf-assertion A* $\wedge$ *wf-assertion B*
| *wf-assertion* (*And A B*) $\longleftrightarrow$ *wf-assertion A* $\wedge$ *wf-assertion B*

| *wf-assertion* (*Wand A B*) ⟷ *wf-assertion B*
| *wf-assertion* (*Imp A B*) ⟷ *pure A* ∧ *wf-assertion B*
| *wf-assertion* (*Wildcard A*) ⟷ *wf-assertion A*
| *wf-assertion* - ⟷ *False*

lemma *wf-implies-combinable*:
assumes *wf-assertion A*
and *sem-combinable* Δ
shows *combinable* Δ *A*
using *assms*
proof (*induct A*)
case (*Exists x A*)
then show *?case*
by (*meson combinable-exists wf-assertion.simps*(*5*))
next
case (*Forall x A*)
then show *?case*
by (*meson combinable-forall wf-assertion.simps*(*4*))
next
case (*Sem B*)
show *?case*
proof (*rule combinableI*)
fix *a b p q x* σ *s*
assume *a*, *s*, Δ ⊨ *Sem B* ∧ *b*, *s*, Δ ⊨ *Sem B* ∧ *Some x* = *p* ⊙ *a* ⊕ *q* ⊙ *b* ∧ *sadd p q* = *one*
then show *x*, *s*, Δ ⊨ *Sem B*
by (*metis Sem.prems*(*1*) *combinable-sem-def sat.simps*(*4*) *wf-assertion.simps*(*2*))
qed
next
case (*Mult x1a A*)
then show *?case*
using *combinable-mult wf-assertion.simps*(*3*) **by** *blast*
next
case (*Star A1 A2*)
then show *?case*
using *combinable-star wf-assertion.simps*(*6*) **by** *blast*
next
case (*Wand A1 A2*)
then show *?case*
using *combinable-wand wf-assertion.simps*(*8*) **by** *blast*
next
case (*And A1 A2*)
then show *?case*
using *combinable-and* **by** *auto*
next
case (*Imp A1 A2*)
then show *?case*

using *combinable-imp* **by** *auto*
next
case *Pred*
show *?case*
proof (*rule combinableI*)
fix *a b p q x σ s*
assume *a*, *s*, Δ ⊨ *Pred* ∧ *b*, *s*, Δ ⊨ *Pred* ∧ *Some x* = *p* ⊙ *a* ⊕ *q* ⊙ *b* ∧ *sadd p q* = *one*
then show *x*, *s*, Δ ⊨ *Pred*
using *assms*(*2*) *sat.simps*(*10*) *sem-combinableE* **by** *metis*
qed
next
case (*Wildcard A*)
then show *?case*
using *combinable-wildcard wf-assertion.simps*(*10*) **by** *blast*
qed (*auto*)

7.4 Theorems

The following two theorems correspond to the rules shown in Section 5.1 of the paper [5].

theorem *apply-wand*:
Star (*syn-mult π A*) (*Mult π* (*Wand A B*)), Δ ⊢ *syn-mult π B*
proof (*rule entailsI*)
fix *σ s*
assume *asm*: *σ*, *s*, Δ ⊨ *Star* (*syn-mult π A*) (*Mult π* (*Wand A B*))
then obtain *x y* **where** *Some σ* = *x* ⊕ *y x*, *s*, Δ ⊨ *syn-mult π A y*, *s*, Δ ⊨ *Mult π* (*Wand A B*)
by *auto*
then have *y*, *s*, Δ ⊨ *Wand* (*syn-mult π A*) (*syn-mult π B*)
by (*metis syn-mult.simps*(*2*) *syn-sen-mult-same*)
then show *σ*, *s*, Δ ⊨ *syn-mult π B*
using ‹*Some σ* = *x* ⊕ *y*› ‹*x*, *s*, Δ ⊨ *syn-mult π A*› ‹*y*, *s*, Δ ⊨ *Wand* (*syn-mult π A*) (*syn-mult π B*)› *commutative* **by** *auto*
qed

theorem *package-wand*:
assumes *Star F* (*syn-mult π A*), Δ ⊢ *syn-mult π B*
shows *F*, Δ ⊢ *Mult π* (*Wand A B*)
by (*metis adjunct2 assms entails-def syn-mult.simps*(*2*) *syn-sen-mult-same*)

The following four theorems correspond to the rules shown in Section 5.2 of the paper [5].

theorem *fold-lfp*:
assumes *pos-neg-rec-call True A*
shows *syn-mult π A*, *LFP* (*applies-eq A*) ⊢ *Mult π Pred*
by (*simp add*: *assms entails-def exists-lfp-gfp*(*1*) *syn-sen-mult-same*)

theorem *unfold-lfp*:
assumes *pos-neg-rec-call True A*
shows *Mult π Pred*, *LFP* (*applies-eq A*) ⊢ *syn-mult π A*
by (*simp add*: *assms entails-def exists-lfp-gfp*(*1*) *syn-sen-mult-same*)

theorem *fold-gfp*:
assumes *pos-neg-rec-call True A*
shows *syn-mult π A*, *GFP* (*applies-eq A*) ⊢ *Mult π Pred*
by (*simp add*: *assms entails-def exists-lfp-gfp*(*2*) *syn-sen-mult-same*)

theorem *unfold-gfp*:
assumes *pos-neg-rec-call True A*
shows *Mult π Pred*, *GFP* (*applies-eq A*) ⊢ *syn-mult π A*
by (*simp add*: *assms entails-def exists-lfp-gfp*(*2*) *syn-sen-mult-same*)

The following theorems correspond to the rule shown in Section 5.3 of the paper [5].

theorem *wf-assertion-combinable-lfp*:
assumes *wf-assertion A*
and *pos-neg-rec-call True A*
shows *sem-combinable* (*LFP* (*applies-eq A*))
proof −
let *?f* = λ*a b*. { *σ* |*σ p q*. *sadd p q* = *one* ∧ *Some σ* = *p* ⊙ *a* ⊕ *q* ⊙ *b*}
have *set-closure-property ?f* (*LFP* (*applies-eq A*))
proof (*rule FP-preserves-set-closure-property*(*2*))
show *monotonic* (*applies-eq A*)
using *assms*(*2*) *pos-neg-rec-call-mono* **by** *blast*
fix Δ :: (*'d*, *'c*, *'a*) *interp* **assume** *asm0*: *set-closure-property ?f* Δ
then have *sem-combinable* Δ
by (*metis combinable-set-closure*)
then show *set-closure-property ?f* (*applies-eq A* Δ)
by (*metis assms*(*1*) *combinable-set-closure sem-combinable-equiv wf-implies-combinable*)
qed
then show *?thesis* **using** *combinable-set-closure* **by** *metis*
qed

theorem *wf-assertion-combinable-gfp*:
assumes *wf-assertion A*
and *pos-neg-rec-call True A*
shows *sem-combinable* (*GFP* (*applies-eq A*))
proof −
let *?f* = λ*a b*. { *σ* |*σ p q*. *sadd p q* = *one* ∧ *Some σ* = *p* ⊙ *a* ⊕ *q* ⊙ *b*}
have *set-closure-property ?f* (*GFP* (*applies-eq A*))
proof (*rule FP-preserves-set-closure-property*(*1*))
show *monotonic* (*applies-eq A*)
using *assms*(*2*) *pos-neg-rec-call-mono* **by** *blast*
fix Δ :: (*'d*, *'c*, *'a*) *interp* **assume** *asm0*: *set-closure-property ?f* Δ
then have *sem-combinable* Δ

by (*metis combinable-set-closure*)
then show *set-closure-property ?f* (*applies-eq A* Δ)
by (*metis assms*(*1*) *combinable-set-closure sem-combinable-equiv wf-implies-combinable*)
qed
then show *?thesis* **using** *combinable-set-closure* **by** *metis*
qed

theorem *wf-combine*:
assumes *wf-assertion A*
and *pos-neg-rec-call True A*
shows *Star* (*Mult* α *Pred*) (*Mult* β *Pred*), *LFP* (*applies-eq A*) ⊢ *Mult* (*sadd* α β) *Pred*
and *Star* (*Mult* α *Pred*) (*Mult* β *Pred*), *GFP* (*applies-eq A*) ⊢ *Mult* (*sadd* α β) *Pred*
apply (*metis assms*(*1*) *assms*(*2*) *logic.combinable-def logic.wf-implies-combinable logic-axioms wf-assertion.simps*(*1*) *wf-assertion-combinable-lfp*)
by (*metis assms*(*1*) *assms*(*2*) *logic.combinable-def logic.wf-implies-combinable logic-axioms wf-assertion.simps*(*1*) *wf-assertion-combinable-gfp*)

end

end

References

[1] S. Blom, S. Darabi, M. Huisman, and W. Oortwijn. The VerCors tool set: Verification of parallel and concurrent software. In N. Polikarpova and S. Schneider, editors, *Integrated Formal Methods*, pages 102–110, Cham, 2017. Springer International Publishing.

[2] R. Bornat, C. Calcagno, P. W. O'Hearn, and M. J. Parkinson. Permission accounting in separation logic. In J. Palsberg and M. Abadi, editors, *Principle of Programming Languages (POPL)*, pages 259–270. ACM, 2005.

[3] J. Boyland. Checking interference with fractional permissions. In R. Cousot, editor, *Static Analysis (SAS)*, pages 55–72, 2003.

[4] J. Brotherston, D. Costa, A. Hobor, and J. Wickerson. Reasoning over permissions regions in concurrent separation logic. In S. K. Lahiri and C. Wang, editors, *Computer Aided Verification (CAV)*, 2020.

[5] T. Dardinier, P. Müller, and A. J. Summers. Fractional resources in unbounded separation logic. *Proc. ACM Program. Lang.*, 6(OOPSLA2), oct 2022.

[6] T. Dardinier, G. Parthasarathy, N. Weeks, P. Müller, and A. J. Summers. Sound automation of magic wands. In S. Shoham and Y. Vizel,

editors, *Computer Aided Verification*, pages 130–151, Cham, 2022. Springer International Publishing.

[7] B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens. VeriFast: A powerful, sound, predictable, fast verifier for C and Java. In M. G. Bobaru, K. Havelund, G. J. Holzmann, and R. Joshi, editors, *NASA Formal Methods (NFM)*, volume 6617 of *Lecture Notes in Computer Science*, pages 41–55. Springer, 2011.

[8] X.-B. Le and A. Hobor. Logical reasoning for disjoint permissions. In A. Ahmed, editor, *European Symposium on Programming (ESOP)*, 2018.

[9] K. R. M. Leino, P. Müller, and J. Smans. Verification of concurrent programs with Chalice. In *Foundations of Security Analysis and Design V*, volume 5705 of *Lecture Notes in Computer Science*, pages 195–222. Springer, 2009.

[10] P. Müller, M. Schwerhoff, and A. J. Summers. Viper: A verification infrastructure for permission-based reasoning. In B. Jobstmann and K. R. M. Leino, editors, *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 9583, pages 41–62. Springer, 2016.

[11] J. C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *Logic in Computer Science (LICS)*, pages 55–74. IEEE, 2002.