A Deciable Fragment of Separation Logic

Florian Sextl

Technical University of Munich

January 9, 2021

Abstract

We present a formalization of the influential paper 'A Decidable Fragment of Separation Logic' by Berdine et al [1]. This formalization follows the original paper in great detail and serves both as a follow up to a seminar paper [3] as well as my submission for the 'Be creative!' homework challenge in our Semantics of Programming Languages course. Another noteworthy followup of the aforemnetioned seminar paper is the proof-of-concept decision procedure Alice_rs [2] which was implemented before this formalization.

The scope for this submission was to prove the UnrollCollapse rule - and as a result also the whole decision procedure - sound and complete. Yet, as of the time I'm writing this, it took so long to prove the other rules that it seems not possible to complete this goal.

Contents

theory *Assertion_Lang*
imports *Main*
begin

1 Syntax and Datatytpes

Defines the language of formulae.

type_synonym *var* = *string*
type_synonym *lval* = *nat*
datatype *val* = *Nilval* | *Val lval*
type_synonym *stack* = *var* ⇒ *val*
type_synonym *heap* = *lval* ⇀ *val*
type_synonym *state* = *stack* × *heap*

datatype *expr* =
  Nil (*nil* 61)
  | *Var var* (´_‘ [0] 61)
datatype *pure* =
  Eq expr expr (_ $=_p$ _ [60, 61] 62)
  | *Neq expr expr* (_ $\neq_p$ _ [60, 61] 62)
type_synonym *pure_form* = *pure list*
datatype *spatial* =
  PointsTo expr expr (_ ⟼ _ [60, 61] 60)
  | *Ls expr* × *expr* (*ls*_ [61] 60)
type_synonym *spatial_form* = *spatial list*
datatype *formula* =
  Pure pure
  | *PureF pure_form*
  | *Spat spatial*
  | *SpatF spatial_form*
  | *Form pure_form spatial_form* (_ | _ [60, 61] 59)

abbreviation *PureTrue* ≡ ([] :: *pure_form*)
notation *PureTrue* (⊤ 61)
abbreviation *pure_conj* ≡ (*Cons* :: *pure* ⇒ *pure_form* ⇒ *pure_form*)
notation *pure_conj* (_ $\wedge_p$ _ [62, 61] 63)
abbreviation *emp* ≡ ([] :: *spatial_form*)
abbreviation *sep_conj* ≡ (*Cons* :: *spatial* ⇒ *spatial_form* ⇒ *spatial_form*)
notation *sep_conj* (_ ∗ _ [60, 61] 61)

term ´*x*‘$\neq_p$´*y*‘ $\wedge_p$ ´*y*‘$\neq_p$*nil* $\wedge_p$ ´*z*‘$=_p$´*x*‘ $\wedge_p$ ⊤ | ´*x*‘ ⟼ ´*y*‘ ∗ *ls*(´*y*‘,*nil*) ∗ *emp*

term $[\acute{}x`\neq_p\acute{}y`,\ \acute{}y`\neq_p nil,\ \acute{}z`=_p\acute{}x`] \mid [\acute{}x` \longmapsto \acute{}y`,\ ls(\acute{}y`,nil)]$
end
theory *Assertion_Misc*
imports *Assertion_Lang*
begin

2 Miscellaneous definitions used with the assertion language

A type class of functions that extract used variables (similarly to vars).

class *fv* =
fixes *fv* :: *'a* ⇒ *var set*

A type class of functions that extract used variables from lists.

class *fvl* = *fv* +
fixes *fvl* :: *'a list* ⇒ *var set*

instantiation *expr* :: *fv*
begin
fun *fv_expr* :: *expr* ⇒ *var set* **where**
fv_expr (*nil*) = {} |
fv_expr (´*v*‘) = {*v*}
instance ..
end

instantiation *pure* :: *fv*
begin
fun *fv_pure* :: *pure* ⇒ *var set* **where**
fv_pure ($e1 =_p e2$) = *fv e1* ∪ *fv e2* |
fv_pure ($e1 \neq_p e2$) = *fv e1* ∪ *fv e2*
instance ..
end

instantiation *pure* :: *fvl*
begin
fun *fvl_pure* :: *pure_form* ⇒ *var set* **where**
fvl_pure pf = $\bigcup$ (*fv*‘(*set pf*))
instance ..
end

instantiation *spatial* :: *fv*
begin

fun *fv_spatial* :: *spatial* ⇒ *var set* **where**
fv_spatial ($e1 \longmapsto e2$) = *fv* $e1$ ∪ *fv* $e2$ |
fv_spatial (*ls*($e1$,$e2$)) = *fv* $e1$ ∪ *fv* $e2$
instance ..
end

instantiation *spatial* :: *fvl*
begin
fun *fvl_spatial* :: *spatial_form* ⇒ *var set* **where**
  fvl_spatial sf = $\bigcup$ (*fv*‘(*set sf*))
instance ..
end

instantiation *formula* :: *fv*
begin
fun *fv_formula* :: *formula* ⇒ *var set* **where**
fv_formula (*Pure p*) = *fv p* |
fv_formula (*PureF pf*) = *fvl pf* |
fv_formula (*Spat s*) = *fv s* |
fv_formula (*SpatF sf*) = *fvl sf* |
fv_formula (*pf* | *sf*) = *fvl pf* ∪ *fvl sf*
instance ..
end

lemma *fv_finite_expr*: *finite* (*fv* (*x*::*expr*))
by (*metis finite.simps fv_expr.elims*)

lemma *fv_finite_un*: $\exists\, v.\ v \notin$ *fv* (*x*::*expr*) ∪ *fv* (*y*::*expr*)
using *fv_finite_expr Finite_Set.finite_Un Finite_Set.ex_new_if_finite infinite_UNIV_listI*
by *metis*

lemma *fv_other_x*: $\bigwedge x.\ \exists\, x'.\ x' \notin$ *fv* (*e*::*expr*) $\wedge\ x' \neq x$
  using *fv_finite_expr fv_finite_un* **by** (*metis Un_iff fv_expr.simps*(2) *insertI*1)

lemma *fv_other_x_un*: $x \notin$ *fv* ($e1$::*expr*) ∪ *fv* ($e2$::*expr*) $\Longrightarrow \exists\, x'.\ x' \notin$ *fv* ($e1$::*expr*) ∪ *fv* ($e2$::*expr*) $\wedge\ x' \neq x$
proof −
assume *assm*: $x \notin$ *fv* ($e1$::*expr*) ∪ *fv* ($e2$::*expr*)
have *finite* (*fv e*1 ∪ *fv e*2) **using** *fv_finite_expr* **by** *simp*
hence ¬*finite*(−(*fv e*1 ∪ *fv e*2)) **by** (*meson finite_compl infinite_UNIV_listI*)
moreover have ¬*finite* (*S*::*string set*) $\Longrightarrow \forall\, y \in S.\ \exists\, y' \in S.\ y{\neq}y'$ **for** *S*
by (*metis* (*no_types*, *hide_lams*) *finite.simps insertCI insert_absorb insert_is_Un subsetI subset_antisym sup_ge*1)

ultimately obtain y **where** $y \in -(fv\ e1 \cup fv\ e2)$ $y{\neq}x$ **using** *assm* **by** *auto*
thus *?thesis* **by** *auto*
qed

Orthogonality property of heaps.

abbreviation *orthogonal* $h1\ h2 \equiv dom\ h1 \cap dom\ h2 = \{\}$
notation *orthogonal* ($\_ \perp \_$ [60, 61] 61)

theorem *ortho_commut*: $h1 \perp h2 \longleftrightarrow h2 \perp h1$ **by** *auto*
theorem *ortho_distr*: $h1 \perp (h2{+}{+}h3) \longleftrightarrow (h1 \perp h2 \wedge h1 \perp h3)$ **by** *auto*

A type class that functions that substitute a variable for another expression.

class *subst* =
fixes *subst* :: $var \Rightarrow expr \Rightarrow 'a \Rightarrow 'a$

A type class that functions that substitute a variable for another expression in a list.

class *substl* = *subst* +
fixes *substl* :: $var \Rightarrow expr \Rightarrow 'a\ list \Rightarrow 'a\ list$

instantiation *expr* :: *subst*
begin
fun *subst_expr* :: $var \Rightarrow expr \Rightarrow expr \Rightarrow expr$ **where**
subst_expr v _ Nil = *Nil* |
subst_expr v e (´*x*‘) = (*if v=x then e else* ´*x*‘)
instance ..
end

instantiation *pure* :: *subst*
begin
fun *subst_pure* :: $var \Rightarrow expr \Rightarrow pure \Rightarrow pure$ **where**
subst_pure v e $(e1 =_p e2) = (subst\ v\ e\ e1) =_p (subst\ v\ e\ e2)$ |
subst_pure v e $(e1 \neq_p e2) = (subst\ v\ e\ e1) \neq_p (subst\ v\ e\ e2)$
instance ..
end

instantiation *pure* :: *substl*
begin
fun *substl_pure* :: $var \Rightarrow expr \Rightarrow pure\_form \Rightarrow pure\_form$ **where**
substl_pure v e = map (subst v e)
instance ..
end

instantiation *spatial* :: *subst*
begin
fun *subst_spatial* :: *var* ⇒ *expr* ⇒ *spatial* ⇒ *spatial* **where**
subst_spatial v e (*e*1 ⟼ *e*2) = (*subst v e e*1) ⟼ (*subst v e e*2) |
subst_spatial v e (*ls*(*e*1, *e*2)) = *ls*((*subst v e e*1), (*subst v e e*2))
instance ..
end

instantiation *spatial* :: *substl*
begin
fun *substl_spatial* :: *var* ⇒ *expr* ⇒ *spatial_form* ⇒ *spatial_form* **where**
substl_spatial v e = *map* (*subst v e*)
instance ..
end

instantiation *formula* :: *subst*
begin
fun *subst_formula* :: *var* ⇒ *expr* ⇒ *formula* ⇒ *formula* **where**
subst_formula v e (*Pure p*) = *Pure* (*subst v e p*) |
subst_formula v e (*PureF pf*) = *PureF* (*substl v e pf*) |
subst_formula v e (*Spat s*) = *Spat* (*subst v e s*) |
subst_formula v e (*SpatF sf*) = *SpatF* (*substl v e sf*) |
subst_formula v e (*pf* | *sf*) = (*substl v e pf*) | (*substl v e sf*)
instance ..
end

lemma *subst_not_free_expr*[*simp*]: *v* ∉ *fv* (*e*::*expr*) ⟹ *subst v E e* = *e*
by (*induction e*) *auto*

lemma *subst_not_eq_expr*[*simp*]: *e*≠´*v*‘ ⟹ *subst v E e* = *e*
by (*induction e*) *auto*

lemma *subst_not_free_pure*[*simp*]: *v* ∉ *fv* (*p*::*pure*) ⟹ *subst v E p* = *p*
by (*induction p*) *auto*

lemma *subst_not_free_spatial*[*simp*]: *v* ∉ *fv* (*s*::*spatial*) ⟹ *subst v E s* = *s*
by (*induction s*) *auto*

lemma *subst_not_free_puref*[*simp*]: *v* ∉ *fvl* (*pf*::*pure_form*) ⟹ *substl v E pf* = *pf*
by (*auto simp*: *map_idI*)

lemma *subst_not_free_spatialf*[*simp*]: *v* ∉ *fvl* (*sf*::*spatial_form*) ⟹ *substl v E sf* = *sf*

by (*auto simp*: *map_idI*)

lemma *subst_not_free_formula*[*simp*]: $x \notin$ *fv* (*F*::*formula*) $\Longrightarrow$ *subst x E F* = *F*
proof (*induction F*)
  case (*PureF x*)
  then show *?case* **using** *subst_not_free_puref* **by** *auto*
next
  case (*SpatF x*)
  then show *?case* **using** *subst_not_free_spatialf* **by** *auto*
next
  case (*Form x1a x2a*)
  then show *?case* **using** *subst_not_free_puref subst_not_free_spatialf* **by** *auto*
qed *simp_all*

lemma *subst_distinct_pure1*: *subst x E P* = $e1 =_p e2 \Longrightarrow \exists\, e3\ e4.\ P = e3 =_p e4$
using *subst_pure.elims* **by** *blast*

lemma *subst_distinct_pure2*: *subst x E P* = $e1 \neq_p e2 \Longrightarrow \exists\, e3\ e4.\ P = e3 \neq_p e4$
using *subst_pure.elims* **by** *blast*

lemma *subst_distinct_puref*: *substl x E Pf* = $P \wedge_p \Pi \Longrightarrow \exists\, P'\ \Pi'.\ Pf = P' \wedge_p \Pi'$
  by *auto*

lemmas *subst_distinct_pure* = *subst_distinct_pure1 subst_distinct_pure2 subst_distinct_puref*

lemma *subst_distinct_spat1*: *subst x E S* = $e1 \longmapsto e2 \Longrightarrow \exists\, e3\ e4.\ S = e3 \longmapsto e4$
using *subst_spatial.elims* **by** *blast*

lemma *subst_distinct_spat2*: *subst x E S* = *ls*(*e1*,*e2*) $\Longrightarrow \exists\, e3\ e4.$ *S* = *ls*(*e3*,*e4*)
using *subst_spatial.elims* **by** *blast*

lemma *subst_distinct_spatf*: *substl x E Sf* = $S{*}\Sigma \Longrightarrow \exists\, S'\ \Sigma'.\ Sf = S'{*}\Sigma'$
  by *auto*

lemmas *subst_distinct_spat* = *subst_distinct_spat1 subst_distinct_spat2 subst_distinct_spatf*

lemma *subst_distinct_formula1*: *subst x E F* = *PureF P* $\Longrightarrow \exists\, P'.$ *F* = *PureF P'*
using *subst_formula.elims* **by** *blast*

lemma *subst_distinct_formula2*: *subst x E F* = *Pure P* $\Longrightarrow \exists P'$. *F* = *Pure* P'
using *subst_formula.elims* **by** *blast*

lemma *subst_distinct_formula3*: *subst x E F* = *SpatF S* $\Longrightarrow \exists S'$. *F* = *SpatF* S'
using *subst_formula.elims* **by** *blast*

lemma *subst_distinct_formula4*: *subst x E F* = *Spat S* $\Longrightarrow \exists S'$. *F* = *Spat* S'
using *subst_formula.elims* **by** *blast*

lemma *subst_distinct_formula5*: *subst x E F* = $\Pi|\Sigma \Longrightarrow \exists \Pi' \Sigma'$. *F* = $\Pi'|\Sigma'$
using *subst_formula.elims* **by** *blast*

lemma *subst_preserve_True[simp]*: *subst x E F* = *PureF* [] $\Longrightarrow$ *F* = *PureF* []
using *subst_distinct_formula1* **by** *fastforce*

lemma *subst_preserve_emp[simp]*: *subst x E F* = *SpatF* [] $\Longrightarrow$ *F* = *SpatF* []
using *subst_distinct_formula3* **by** *fastforce*

lemmas *subst_distinct_formula* = *subst_distinct_formula1* *subst_distinct_formula2*
subst_distinct_formula3 *subst_distinct_formula4* *subst_distinct_formula5*
subst_preserve_True *subst_preserve_emp*

lemma *subst_reflexive*: ´*x*‘=*E* $\Longrightarrow$ *subst x E* (*e*::*expr*) = *e*
using *subst_expr.elims* **by** *metis*

lemma *subst_fv_expr*: ´*x*‘≠*E* $\Longrightarrow$ *x* $\notin$ *fv* (*subst x E* (*e*::*expr*))
by (*metis empty_iff expr.exhaust fv_expr.simps*(1) *fv_expr.simps*(2) *insert_iff*
subst_expr.simps(2) *subst_not_eq_expr*)

lemma *subst_fv_expr_set*: ´*x*‘≠*E* $\Longrightarrow$ *fv* (*subst x E* (*e*::*expr*)) $\supseteq$ (*fv e* − {*x*})
using *subst_fv_expr* **by** (*metis Diff_cancel Diff_subset fv_expr.simps*(2) *subst_not_eq_expr*)

lemma *subst_fv_expr_set_un*:
´*x*‘≠*E* $\Longrightarrow$ *fv* (*subst x E* (*e1*::*expr*)) $\cup$ *fv* (*subst x E* (*e2*::*expr*)) $\supseteq$ (*fv e1* $\cup$ *fv e2*) − {*x*}
using *subst_fv_expr_set* **by** *auto*

end
theory *Assertion_Semantics*
imports *Assertion_Lang Assertion_Misc*
begin

3 Semantics

Defines the syntax for the assertion language formulae.

3.1 Satisfaction predicate

Satisfactions describe the semantics of the assertion language.

fun $eval :: expr \Rightarrow stack \Rightarrow val$ **where**
$eval\ (nil)\ s = Nilval\ |$
$eval\ (\acute{}x`)\ s = s\ x$
notation $eval$ ($[\![\_]\!]\_$ [60, 61] 61)

A satisfaction with a ls segment holds iff there exists a path of heap cells that point to each other and that form a super list of the given segment.

inductive $ls\_ind :: state \Rightarrow nat \Rightarrow (expr \times expr) \Rightarrow bool$ ($\_\models ls^{\_}\_$ 50) **where**
$EmptyLs$: $[\![e1]\!]s = [\![e2]\!]s \Longrightarrow dom\ h = \{\} \Longrightarrow (s,h)\models ls^0(e1,e2)\ |$
$ListSegment$: $[\![e1]\!]s = Val\ v' \Longrightarrow h1 = [v' \mapsto v] \Longrightarrow xs \subseteq -(fv\ e1 \cup fv\ e2)$
$\Longrightarrow (\forall x \in xs.\ ((s(x:=v),h2)\models ls^m(\acute{}x`,e2))) \Longrightarrow h1 \perp h2 \Longrightarrow h = h1{+}{+}h2$
$\Longrightarrow n = Suc\ m$
$\Longrightarrow [\![e1]\!]s \neq [\![e2]\!]s \Longrightarrow (s,h)\models ls^n(e1,e2)$

inductive $satisfaction :: state \Rightarrow formula \Rightarrow bool$ (**infix** $\models$ 50) **where**
$EqSat$: $[\![e1]\!]s{=}[\![e2]\!]s \Longrightarrow (s,h)\models Pure(e1{=_p}e2)\ |$
$NeqSat$: $[\![e1]\!]s{\neq}[\![e2]\!]s \Longrightarrow (s,h)\models Pure(e1{\neq_p}e2)\ |$
$TrueSat$: $(s,h)\models PureF\ []\ |$
$ConjSat$: $(s,h)\models Pure\ P \Longrightarrow (s,h)\models PureF\ \Pi \Longrightarrow (s,h)\models PureF(P \wedge_p \Pi)\ |$
$PointsToSat$: $[\![[\![e1]\!]s = Val\ v;\ h = [v{\mapsto}[\![e2]\!]s]]\!] \Longrightarrow (s,h)\models Spat(e1 \longmapsto e2)\ |$
$EmpSat$: $h = Map.empty \Longrightarrow (s,h)\models SpatF\ emp\ |$
$SepConjSat$: $h1 \perp h2 \Longrightarrow h = h1{+}{+}h2 \Longrightarrow (s,h1)\models Spat\ S \Longrightarrow (s,h2)\models SpatF$
Σ
$\Longrightarrow (s,h)\models SpatF(S * \Sigma)\ |$
$FormSat$: $(s,h)\models PureF\ \Pi \Longrightarrow (s,h)\models SpatF\ \Sigma \Longrightarrow (s,h)\models(\Pi\ |\ \Sigma)\ |$
$LsSat$: $(s,h)\models ls^n(e1,e2) \Longrightarrow (s,h)\models Spat(ls(e1,e2))$

declare *ls_ind.intros*[*intro*]
declare *satisfaction.intros*[*intro*]

lemmas *ls_induct* = *ls_ind.induct*[*split_format*(*complete*)]
lemmas *sat_induct* = *satisfaction.induct*[*split_format*(*complete*)]

inductive_cases [*elim*]: $(s,h) \models ls^0(e1,e2)$ $(s,h) \models ls^n(e1,e2)$
inductive_cases [*elim*]: $(s,h) \models Pure(e1 =_p e2)$ $(s,h) \models Pure(e1 \neq_p e2)$ $(s,h) \models PureF$ []
$(s,f) \models PureF(P \wedge_p \Pi)$ $(s,h) \models Spat(e1 \longmapsto e2)$ $(s,h) \models SpatF\ emp$ $(s,h) \models SpatF(S * \Sigma)$
$(s,h) \models (\Pi \mid \Sigma)$ $(s,h) \models Spat(ls(e1,e2))$

3.2 Satisfaction properties

There are a number of helpful properties that follow from the satisfaction definition.

Satisfaction is decidable, cf. Lemma 1 [1].

corollary *sat_decidable*: $(s,h) \models F \vee \neg\ (s,h) \models F$
by *simp*

Separating conjunctions are only allowed on distinct heap parts.

corollary *sep_conj_ortho*: $\nexists s\ h.\ (s,h) \models [\acute{}x`{=_p}\acute{}y`] \mid [\acute{}x` \longmapsto xv,\ \acute{}y` \longmapsto yv]$
proof
assume $\exists s\ h.\ (s,\ h) \models [\acute{}x` =_p\ \acute{}y`] \mid [\acute{}x` \longmapsto xv,\ \acute{}y` \longmapsto yv]$
then obtain $s\ h$ **where** $(s,h) \models [\acute{}x`{=_p}\acute{}y`] \mid [\acute{}x` \longmapsto xv,\ \acute{}y` \longmapsto yv]$ **by** *auto*
hence $(s,h) \models PureF\ [\acute{}x`{=_p}\acute{}y`]$ **and** *spatf*: $(s,h) \models SpatF\ [\acute{}x` \longmapsto xv,\ \acute{}y` \longmapsto yv]$ **by** *auto*
{
hence $(s,h) \models Pure(\acute{}x`{=_p}\acute{}y`)$ **by** *auto*
hence $[\![\acute{}x`]\!]s = [\![\acute{}y`]\!]s$ **by** *auto*
}
from *spatf* **obtain** $h1\ h2$ **where** $h1 \perp h2$ $h = h1{+}{+}h2$ $(s,h1) \models Spat\ (\acute{}x`{\longmapsto}xv)$
$(s,h2) \models SpatF\ [\acute{}y`{\longmapsto}yv]$ **by** *blast*
from $\langle(s,h1) \models Spat\ (\acute{}x`{\longmapsto}xv)\rangle$ **obtain** $v$ **where** $[\![\acute{}x`]\!]s = Val\ v$ $dom\ h1 = \{v\}$ **by** *auto*
from $\langle(s,h2) \models SpatF\ [\acute{}y`{\longmapsto}yv]\rangle$ **obtain** $h3\ h4$ **where** $h3 \perp h4$ $h2 = h3{+}{+}h4$
$(s,h3) \models Spat(\acute{}y`{\longmapsto}yv)$ **by** *auto*
then obtain v' **where** $[\![\acute{}y`]\!]s = Val\ v'$ $dom\ h3 = \{v'\}$ **by** *auto*
with $\langle[\![\acute{}x`]\!]s = Val\ v\rangle$ $\langle[\![\acute{}x`]\!]s = [\![\acute{}y`]\!]s\rangle$ **have** $v = v'$ **by** *simp*
with $\langle dom\ h3 = \{v'\}\rangle$ $\langle dom\ h1 = \{v\}\rangle$ **have** $\neg\ h3 \perp h1$ **by** *simp*
hence $\neg\ (h3{+}{+}h4) \perp h1$ **using** *ortho_distr* **by** *auto*

with ⟨$h2 = h3{+}{+}h4$⟩ **have** $\neg\ h1 \perp h2$ **using** *ortho_commut* **by** *metis*
with ⟨$h1 \perp h2$⟩ **show** *False* **by** *simp*
qed

Order in pure formulae does not matter.

corollary *pure_commut*: $(s,h) \models PureF(p1 \wedge_p p2 \wedge_p \Pi) \longleftrightarrow (s,h) \models PureF(p2 \wedge_p p1 \wedge_p \Pi)$
by *auto*
corollary *pure_commut_form*: $(s,h) \models (p1 \wedge_p p2 \wedge_p \Pi) | \Sigma \Longrightarrow (s,h) \models (p2 \wedge_p p1 \wedge_p \Pi) | \Sigma$
using *pure_commut* **by** *force*

Singular spatial formulae are only satisfied by singular heaps.

corollary *sing_heap*: $(s,h) \models SpatF[x \longmapsto y] \longleftrightarrow (s,h) \models Spat(x \longmapsto y) \wedge (\exists\ v\ v'.\ [\![x]\!]s = Val\ v \wedge$
$[\![y]\!]s = v' \wedge h = [v \mapsto v'])$ (**is** *?lhs* $\longleftrightarrow$ *?rhs*)
proof
assume *?lhs*
hence *spat*: $(s,\ h) \models Spat\ (x \longmapsto y)$ **by** *fastforce*
moreover then obtain $v\ v'$ **where** $[\![x]\!]s = Val\ v\ [\![y]\!]s = v'$ **by** *blast*
moreover with *spat* **have** $h = [v \mapsto v']$ **by** *fastforce*
ultimately show *?rhs* **by** *simp*
next
assume *?rhs*
moreover have $h \perp Map.empty$ **by** *simp*
ultimately have $(s,h{+}{+}Map.empty) \models SpatF[x \longmapsto y]$ **by** *blast*
thus *?lhs* **by** *simp*
qed

Order in spatial formulae does not matter.

corollary *spatial_commut*: $(s,h) \models SpatF(s1 * s2 * \Sigma) \longleftrightarrow (s,h) \models SpatF(s2 * s1 * \Sigma)$
(**is** *?P s1 s2* $\longleftrightarrow$ *?p s2 s1*)
proof
assume *?P s1 s2*
then obtain $h1\ h2$ **where** $h{:}h1 \perp h2 \wedge h = h1{+}{+}h2$ **and** $s1{:}(s,h1) \models Spat\ s1$ **and** $(s,h2) \models SpatF\ (s2 * \Sigma)$
by *auto*
then obtain $h3\ h4$ **where** $h2{:}h3 \perp h4 \wedge h2 = h3{+}{+}h4$ **and** $s2{:}\ (s,h3) \models Spat\ s2$ **and** $\sigma{:}\ (s,h4) \models SpatF\ \Sigma$
by *auto*
from $h\ h2$ **have** $h4 \perp h1$ **by** *auto*
moreover then obtain $h2'$ **where** $h2'{:}\ h2' = h1{+}{+}h4$ **by** *simp*
ultimately have $(s,h2') \models SpatF(s1 * \Sigma)$ **using** $s1\ \sigma$ **by** *auto*
moreover from $h\ h2\ h2'$ **have** $h3 \perp h2'$ **by** *auto*
moreover with $h\ h2\ h2'$ **have** $h{=}h3{+}{+}h2'$ **by** (*metis map_add_assoc map_add_comm*)

ultimately show *?P s2 s1* **using** *s2* **by** *auto*
next
assume *?P s2 s1*
then obtain *h1 h2* **where** *h*:$h1 \perp h2 \land h = h1{+}{+}h2$ **and** *s2*:$(s,h1) \models$ *Spat s2* **and** $(s,h2) \models$ *SpatF* $(s1{*}\Sigma)$
by *auto*
then obtain *h3 h4* **where** *h2*:$h3 \perp h4 \land h2 = h3{+}{+}h4$ **and** *s1*:$(s,h3) \models$ *Spat s1* **and** σ: $(s,h4) \models$ *SpatF* Σ
by *auto*
from *h h2* **have** $h4 \perp h1$ **by** *auto*
moreover then obtain $h2'$ **where** $h2'$: $h2' = h1{+}{+}h4$ **by** *simp*
ultimately have $(s,h2') \models$ *SpatF*$(s2{*}\Sigma)$ **using** *s2* σ **by** *auto*
moreover from *h h2* $h2'$ **have** $h3 \perp h2'$ **by** *auto*
moreover with *h h2* $h2'$ **have** $h{=}h3{+}{+}h2'$ **by** (*metis map_add_assoc map_add_comm*)
ultimately show *?P s1 s2* **using** *s1* **by** *auto*
qed
corollary *spatial_commut_form*: $(s,h) \models \Pi|(s1{*}s2{*}\Sigma) \Longrightarrow (s,h) \models \Pi|(s2{*}s1{*}\Sigma)$
using *spatial_commut* **by** *force*

An empty list is equivalent to an empty heap.

corollary *empty_ls*: $(s,h) \models$ *SpatF emp* $\longleftrightarrow (s,h) \models$ *Spat*(*ls*(x,x))
proof
assume $(s,\ h) \models$ *SpatF emp*
hence *dom* $h = \{\}$ **by** *blast*
hence $(s,h) \models ls^0(x,x)$ **by** *blast*
thus $(s,\ h) \models$ *Spat* (*ls*$(x,\ x)$) **by** *blast*
next
assume $(s,\ h) \models$ *Spat* (*ls*$(x,\ x)$)
then obtain n **where** $(s,h) \models ls^n(x,x)$ **by** *auto*
hence $n{=}0$ *dom* $h = \{\}$ **by** *auto*
thus $(s,\ h) \models$ *SpatF emp* **by** *auto*
qed

Due to this theorem circular list segements can only be formulated as follows:

term *´x‘*$\longmapsto$*´y‘* $*$ *ls*(*´y‘*,*´y‘*) $*$ *emp*

The heap has no influence on the satisfaction of a pure formula.

corollary *heap_pure*: $(s,h) \models$ *Pure P* $\Longrightarrow \forall\, h'.\ (s,h') \models$ *Pure P*
by (*induction s h Pure P rule*: *sat_induct*) *auto*
corollary *heap_puref*: $(s,h) \models$ *PureF* $\Pi \Longrightarrow \forall\, h'.\ (s,h') \models$ *PureF* Π
proof (*induction s h PureF* Π *arbitrary*: Π *rule*: *sat_induct*)
case (*TrueSat s h*)
then show *?case* **by** *fast*

next
  case (*ConjSat s h P* Π')
  then show *?case* **using** *heap_pure* **by** *blast*
qed

Evaluation does not rely on unrelated variable values.

corollary *eval_notin*[*simp*]: $x \notin fv\ e \Longrightarrow [\![e]\!]s{=}[\![e]\!]s(x{:=}v)$
by (*cases e*) *auto*

Only the two ls expressions are stack related.

corollary *ls_stack_relation*: $[\![(s,h){\models}ls^n(e1,e2);\ [\![e1]\!]s{=}[\![e1]\!]t;\ [\![e2]\!]s{=}[\![e2]\!]t]\!] \Longrightarrow (t,h){\models}ls^n(e1,e2)$
proof (*induction arbitrary*: *t rule*: *ls_induct*)
  case (*EmptyLs e*1 *s e*2 *h*)
  then show *?case* **by** *auto*
next
  case (*ListSegment e*1 *s* v' *h*1 *v xs e*2 *h*2 *m h n*)
  from *ListSegment.hyps*(1) *ListSegment.prems*(1) **have** *e*1: $[\![e1]\!]t = Val\ v'$ **by** *simp*
  from *ListSegment.hyps*(7) *ListSegment.prems* **have** *neq*: $[\![e1]\!]t \neq [\![e2]\!]t$ **by** *simp*
  have $\forall x \in xs.\ (t(x{:=}v),h2){\models}ls^m(\acute{}x`,\ e2)$
  proof
    fix *x* :: *var*
    assume *assm*: $x \in xs$
    with *ListSegment.IH* **have** *aux*:
      $[\![\acute{}x`]\!]s(x := v) = [\![\acute{}x`]\!]xa \Longrightarrow [\![e2]\!]s(x := v) = [\![e2]\!]xa \Longrightarrow (xa, h2){\models}ls^m(\acute{}x`,\ e2)$ **for** *xa*
      by *blast*
    have $[\![\acute{}x`]\!]s(x{:=}v) = [\![\acute{}x`]\!]t(x{:=}v)$ $[\![e2]\!]s(x{:=}v) = [\![e2]\!]t(x{:=}v)$ **using** *assm ListSegment.prems*
      apply *simp* **using** *assm ListSegment.prems ListSegment*(3)
      by (*metis ComplD UnCI eval_notin subsetD*)
    from *aux*[*OF this*] **show** $(t(x{:=}v),h2){\models}ls^m(\acute{}x`,\ e2)$ **.**
  qed
  from *ls_ind.ListSegment*[*OF e*1 *ListSegment*(2−3) *this ListSegment*(4−6) *neq*] **show** *?case* **.**
qed

lemma *ls_extend_lhs*: $[\![(s(x{:=}v),h){\models}ls^n(e1,e2);\ x \notin fv\ e1 \cup fv\ e2]\!] \Longrightarrow (s,h){\models}ls^n(e1,e2)$
proof −
  assume *assm*1: $(s(x{:=}v),h){\models}ls^n(e1,e2)$
  assume *assm*2: $x \notin fv\ e1 \cup fv\ e2$
  hence $[\![e1]\!]s(x{:=}v) = [\![e1]\!]s$ $[\![e2]\!]s(x{:=}v) = [\![e2]\!]s$ **using** *eval_notin* **by**

fastforce+
  from *ls_stack_relation*[*OF assm*1 *this*] **show** *?thesis* **.**
qed

lemma *ls_extend_rhs*: $[\![(s,h) \models ls^n(e1,e2);\ x \notin fv\ e1 \cup fv\ e2]\!] \Longrightarrow (s(x{:=}v),h) \models ls^n(e1,e2)$
proof $-$
  assume *assm*1: $(s,h) \models ls^n(e1,e2)$
  assume *assm*2: $x \notin fv\ e1 \cup fv\ e2$
  hence $[\![e1]\!]s = [\![e1]\!]s(x{:=}v)$ $[\![e2]\!]s = [\![e2]\!]s(x{:=}v)$ **using** *eval_notin* **by** *fastforce+*
  from *ls_stack_relation*[*OF assm*1 *this*] **show** *?thesis* **.**
qed

corollary *ls_extend*: $x \notin fv\ e1 \cup fv\ e2 \Longrightarrow ((s,h) \models ls^n(e1,e2)) = ((s(x{:=}v),h) \models ls^n(e1,e2))$
  using *ls_extend_lhs ls_extend_rhs* **by** *metis*

The following lemmata are used to proof the substitution rule:

lemma *subst_expr*: $[\![\acute{}x`]\!]s = [\![E]\!]s \Longrightarrow [\![subst\ x\ E\ e]\!]s = [\![e]\!]s$
using *subst_expr.elims* **by** *metis*

lemma *ls_change_fst*: $[\![(s,h) \models ls^n(a,e);\ [\![a]\!]s = [\![b]\!]s]\!] \Longrightarrow (s,h) \models ls^n(b,e)$
proof (*induction rule*: *ls_induct*)
  case (*EmptyLs e*1 *s e*2 *h*)
  then show *?case* **by** *auto*
next
  case (*ListSegment a s v h*1 *v′ xs e h*2 *h m n*)
  hence *b*: $[\![b]\!]s = Val\ v$ **by** *metis*
  define *xs′* **where** *xs′*: $xs' = xs - fv\ b$
  with *ListSegment*(3) **have** $xs' \subseteq -(fv\ a \cup fv\ e) - fv\ b$ **by** *auto*
  hence *xs′_sub*: $xs' \subseteq -(fv\ b \cup fv\ e)$ **by** *auto*
  have *ih*: $\forall x \in xs'.\ (s(x := v'),\ h2) \models ls^h(\acute{}x`,\ e)$
  proof
    fix x
    assume $x \in xs'$
    with *xs′* **have** $x \in xs$ **by** *simp*
    thus $(s(x := v'),\ h2) \models ls^h(\acute{}x`,\ e)$ **using** *ListSegment.IH* **by** *simp*
  qed
  from *ListSegment.prems*(1) *ListSegment.hyps*(7) **have** $[\![b]\!]s \neq [\![e]\!]s$ **by** *simp*
  from *ls_ind.ListSegment*[*OF b ListSegment.hyps*(2) *xs′_sub ih ListSegment.hyps*(4−6) *this*] **show** *?case* **.**
qed

lemma *ls_change_snd*: $[\![(s,h) \models ls^n(e,a);\ [\![a]\!]s = [\![b]\!]s]\!] \Longrightarrow (s,h) \models ls^n(e,b)$

proof (*induction rule*: *ls_induct*)
  case (*EmptyLs* $e1'$ *s* $e2'$ *h*)
  then show *?case* **by** *auto*
next
  case (*ListSegment* *e s v* $h1$ v' *xs a* $h2$ *h m n*)
  define xs' **where** xs': $xs' = xs - fv\ b$
  with *ListSegment*(3) **have** $xs' \subseteq -\ (fv\ e \cup fv\ a) - fv\ b$ **by** *auto*
  hence xs'*_sub*: $xs' \subseteq -\ (fv\ e \cup fv\ b)$ **by** *auto*
  have *ih*: $\forall x \in xs'.\ (s(x := v'),\ h2) \models ls^h(\acute{}x`,\ b)$
  proof
    fix x
    assume x: $x \in xs'$
    with xs' **have** $x \notin fv\ b$ **by** *simp*
    moreover from x xs' *ListSegment*(3) **have** $x \notin fv\ a$ **by** *auto*
    ultimately have $[\![b]\!]s(x := v') = [\![b]\!]s$ $[\![a]\!]s(x := v') = [\![a]\!]s$ **using** *eval_notin* **by** *metis+*
    with *ListSegment.prems* **have** $[\![a]\!]s(x := v') = [\![b]\!]s(x := v')$ **by** *simp*
    from x xs' **have** $x \in xs$ **by** *simp*
    hence $[\![a]\!]s(x := v') = [\![b]\!]s(x := v') \Longrightarrow (s(x := v'),\ h2) \models ls^h(\acute{}x`,\ b)$ **using** *ListSegment.IH*
      by *blast*
      from *this*[*OF* ‹$[\![a]\!]s(x := v') = [\![b]\!]s(x := v')$›] **show** $(s(x := v'),\ h2) \models ls^h(\acute{}x`,\ b)$ **.**
  qed
  from *ListSegment.prems* *ListSegment.hyps*(7) **have** $[\![e]\!]s \neq [\![b]\!]s$ **by** *simp*
  from *ls_ind.ListSegment*[*OF* *ListSegment*(1−2) xs'*_sub* *ih* *ListSegment*(4−6) *this*] **show** *?case* **.**
qed

lemma *subst_sat_ls*: $[\![(s,h) \models ls^n(e1',e2');\ e1' = subst\ x\ E\ e1;\ e2' = subst\ x\ E\ e2;\ [\![\acute{}x`]\!]s = [\![E]\!]s]\!]$
  $\Longrightarrow (s,h) \models ls^n(e1,e2)$
  using *ls_change_snd ls_change_fst subst_expr* **by** *metis*

lemma *subst_sat*: $[\![(s,h) \models F';\ F' = subst\ x\ E\ F;\ [\![\acute{}x`]\!]s = [\![E]\!]s]\!] \Longrightarrow (s,h) \models F$
proof (*induction arbitrary*: *F rule*: *sat_induct*)
  case (*EqSat* $e1$ *s* $e2$ *h*)
  from *EqSat.prems*(1) **obtain** $e3$ $e4$ **where** *F*: $F = Pure\ (e3 =_p e4)$
    using *subst_distinct_pure*1 *subst_distinct_formula*2
    by (*metis formula.inject*(1) *subst_formula.simps*(1))
  with *EqSat.prems*(1) **have** $e1$: $e1 = subst\ x\ E\ e3$ **and** $e2$: $e2 = subst\ x\ E\ e4$ **by** *simp_all*
  then show *?case* **proof** (*cases* $\acute{}x` = e3$)
    case *True*

with $e1$ **have** $e1 = E$ **by** *auto*
then show *?thesis* **using** *EqSat* **by** (*metis F True e2 satisfaction.EqSat subst_not_eq_expr*)
next
case *False*
then show *?thesis* **proof** (*cases* ´*x*‘=*e4*)
case *True*
with $e2$ **have** $e2 = E$ **by** *auto*
then show *?thesis* **using** *EqSat F False True* **by** *auto*
next
case *False*
with ‹´*x*‘≠*e3*› *F* **have** $x \notin fv\ F$ **by** (*metis Un_iff empty_iff fv_expr.simps*(1) *fv_expr.simps*(2)
fv_formula.simps(1) *fv_pure.simps*(1) *insert_iff subst_expr.elims*)
then show *?thesis* **using** *subst_not_free_formula EqSat.hyps F e1 e2* **by** *auto*
qed
qed
next
case (*NeqSat e1 s e2 h*)
from *NeqSat.prems*(1) **obtain** *e3 e4* **where** *F*: $F = Pure\ (e3 \neq_p e4)$
using *subst_distinct_pure2 subst_distinct_formula2*
by (*metis formula.inject*(1) *subst_formula.simps*(1))
with *NeqSat.prems*(1) **have** *e1*: $e1 = subst\ x\ E\ e3$ **and** *e2*: $e2 = subst\ x\ E\ e4$ **by** *simp_all*
then show *?case* **proof** (*cases* ´*x*‘=*e3*)
case *True*
with $e1$ **have** $e1 = E$ **by** *auto*
then show *?thesis* **using** *NeqSat* **by** (*metis F True e1 e2 satisfaction.NeqSat subst_not_eq_expr*)
next
case *False*
then show *?thesis* **proof** (*cases* ´*x*‘=*e4*)
case *True*
with $e2$ **have** $e2 = E$ **by** *auto*
then show *?thesis* **using** *NeqSat F False True* **by** *auto*
next
case *False*
with ‹´*x*‘≠*e3*› *F* **have** $x \notin fv\ F$ **by** (*smt Un_iff fv_expr.simps*(1) *fv_expr.simps*(2)
fv_formula.simps(1) *fv_pure.simps*(2) *insert_absorb insert_iff insert_not_empty*
subst_expr.elims)
then show *?thesis* **using** *subst_not_free_formula NeqSat.hyps F e1 e2*

```
by auto
    qed
  qed
next
  case (TrueSat s h)
 then show ?case using subst_preserve_True satisfaction.TrueSat by metis
next
  case (ConjSat s h P Π)
  from ConjSat.prems(1) obtain P' Π' where F: F = PureF (P'∧pΠ')
    using subst_distinct_formula1 subst_distinct_puref
    by (metis formula.inject(2) subst_formula.simps(2))
  with ConjSat.prems(1) have Pure P = subst x E (Pure P') PureF Π =
subst x E (PureF Π')
    by simp_all
   from ConjSat.IH(1)[OF this(1) ConjSat.prems(2)] ConjSat.IH(2)[OF
this(2) ConjSat.prems(2)] F
  show ?case by auto
next
  case (PointsToSat e1 s v h e2)
  then show ?case proof (cases x ∈ fv F)
    case True
    then show ?thesis using PointsToSat
   by (smt formula.inject(3) satisfaction.PointsToSat spatial.inject(1) subst_distinct_formula4

     subst_distinct_spat1 subst_expr.simps(2) subst_formula.simps(3) subst_not_eq_expr

       subst_spatial.simps(1))
  next
    case False
     show ?thesis using subst_not_free_formula[OF False] PointsToSat by
fastforce
  qed
next
  case (EmpSat h s)
 then show ?case using satisfaction.EmpSat subst_preserve_emp by metis
next
  case (SepConjSat h1 h2 h s S Σ)
  from SepConjSat.prems(1) obtain S' Σ' where F: F = SpatF (S'∗Σ')
    using subst_distinct_spatf subst_distinct_formula3
    by (metis formula.inject(4) subst_formula.simps(4))
   with SepConjSat.prems(1) have Spat S = subst x E (Spat S') SpatF Σ
= subst x E (SpatF Σ')
    by simp_all
 with SepConjSat.IH SepConjSat.prems(2) have (s,h1)⊨Spat S' (s,h2)⊨SpatF
```

Σ' **by** *simp_all*
then show *?case* **using** *F SepConjSat.hyps* **by** *blast*
next
case (*FormSat s h* Π Σ)
from *FormSat.prems*(1) **obtain** Π' Σ' **where** *F*:$F = \Pi'|\Sigma'$ **using** *subst_distinct_formula5*
by *metis*
with *FormSat.prems*(1) **have** *substl x E* $\Pi' = \Pi$ **and** *substl x E* $\Sigma' = \Sigma$
by *simp_all*
with *FormSat.IH FormSat.prems*(2) **have** $(s,h)\models$*PureF* Π' **and** $(s,h)\models$*SpatF*
Σ' **by** *simp_all*
with *F* **show** *?case* **by** *auto*
next
case (*LsSat s h n e1 e2*)
from *LsSat*(2) **obtain** *e1′ e2′* **where** *F*:*F* = *Spat* (*ls*(*e1′*, *e2′*)) **using**
subst_distinct_formula4
by (*metis formula.inject*(3) *subst_distinct_spat2 subst_formula.simps*(3))
with *LsSat*(2) **have** *e1* = *subst x E e1′ e2* = *subst x E e2′* **by** *simp_all*
from *subst_sat_ls*[*OF LsSat*(1) *this LsSat*(3)] **show** *?case* **using** *F* **by**
auto
qed

lemma *subst_sat_ls_rev*: $[\![(s,h)\models ls^n(e1',e2'); e1 = subst\ x\ E\ e1'; e2 = subst$
$x\ E\ e2'; [\![\acute{}x`]\!]s=[\![E]\!]s]\!]$
$\Longrightarrow (s,h)\models ls^n(e1,e2)$
using *ls_change_snd ls_change_fst subst_expr* **by** *metis*

lemma *subst_sat_rev*:$[\![(s,h)\models F; [\![\acute{}x`]\!]s=[\![E]\!]s]\!] \Longrightarrow (s,h)\models subst\ x\ E\ F$
proof (*induction rule*: *sat_induct*)
case (*EqSat e1 s e2 h*)
then show *?case* **by** (*metis satisfaction.EqSat subst_expr.simps*(2) *subst_formula.simps*(1)
subst_not_eq_expr subst_pure.simps(1))
next
case (*NeqSat e1 s e2 h*)
then show *?case* **by** (*metis satisfaction.NeqSat subst_expr.simps*(2) *subst_formula.simps*(1)
subst_not_eq_expr subst_pure.simps(2))
next
case (*PointsToSat e1 s v h e2*)
then show *?case* **by** (*smt satisfaction.PointsToSat subst_expr.simps*(2)
subst_formula.simps(3)
subst_not_eq_expr subst_spatial.simps(1))
next
case (*LsSat s h n e1 e2*)
obtain *e1′ e2′* **where** *F*: *subst x E* (*Spat* (*ls*(*e1*, *e2*))) = *Spat* (*ls*(*e1′*, *e2′*))
by *simp*

hence $e1' = subst\ x\ E\ e1\ \ e2' = subst\ x\ E\ e2$ **by** *simp_all*
from *subst_sat_ls_rev*[*OF LsSat*(1) *this LsSat*(2)] **show** *?case* **using** *F* **by** *auto*
qed *auto*

lemma *subst_sat_eq*: $[\![F'{=}subst\ x\ E\ F;\ [\![{'x'}]\!]s{=}[\![E]\!]s]\!] \Longrightarrow ((s,h){\models}F') = ((s,h){\models}F)$
using *subst_sat subst_sat_rev* **by** *fast*

end
theory *Entailment*
imports *Assertion_Semantics*
begin

4 Entailments

Entailments formalize single deduction steps in separation logic.

An entailment describes that the consequent is satisfied by at least all states that also satisfy the antecedent.

definition *entails* :: *formula* ⇒ *formula* ⇒ *bool* (**infix** ⊢ 50) **where**
$antecedent \vdash consequent \equiv (\forall\ s\ h.\ (s,h){\models}antecedent \longrightarrow (s,h){\models}consequent)$

Auxiliary lemma to lift reasoning from Isabelle/HOL to Isabelle/Pure

lemma *entailment_lift*: $(\bigwedge s\ h.\ (s,h){\models}\Pi\Sigma \Longrightarrow (s,h){\models}\Pi\Sigma') \Longrightarrow \Pi\Sigma \vdash \Pi\Sigma'$
unfolding *entails_def* **using** *HOL.impI HOL.allI* **by** *simp*

lemma *entailment_lift_rev*: $\Pi\Sigma \vdash \Pi\Sigma' \Longrightarrow (\bigwedge s\ h.\ (s,h){\models}\Pi\Sigma \Longrightarrow (s,h){\models}\Pi\Sigma')$
unfolding *entails_def* **using** *HOL.impI HOL.allI* **by** *simp*

lemma *entailment_trans*: $[\![\Pi\Sigma \vdash \Pi\Sigma';\ \Pi\Sigma' \vdash \Pi\Sigma'']\!] \Longrightarrow \Pi\Sigma \vdash \Pi\Sigma''$
by (*simp add*: *entails_def*)

4.1 Example entailments from the paper

Entailments are reflexive with regard to equality.

lemma *eq_refl*: $[{'x'}{=}_p{'y'},\ E{=}_pF] \mid [{'x'}{\longmapsto}E] \vdash Spat\ ({'y'}{\longmapsto}F)$
proof(*rule entailment_lift*)
fix *s h*
assume *antecedent*: $(s,h){\models}[{'x'}{=}_p{'y'},\ E{=}_pF] \mid [{'x'}{\longmapsto}E]$
hence $(s,h){\models}\ PureF\ [{'x'}{=}_p{'y'},\ E{=}_pF]$ **by** *auto*
hence $[\![{'x'}]\!]s = [\![{'y'}]\!]s\ \ [\![E]\!]s = [\![F]\!]s$ **by** *blast+*

moreover from *sing_heap antecedent* **have** $(s,h) \models Spat\ (\acute{}x`\longmapsto E)$ **by** *auto*
ultimately show $(s,h) \models Spat\ (\acute{}y`\longmapsto F)$ **by** *fastforce*
qed

In the following some simple entailments for list segment are shown to hold.

lemma *emp_entails_ls*: $[x{=}_p y] \mid emp \vdash Spat\ (ls(x,y))$
proof (*rule entailment_lift*)
fix *s h*
assume $(s,\ h) \models [x{=}_p y] \mid emp$
hence $[\![x]\!]s = [\![y]\!]s\ dom\ h = \{\}$ **by** *auto*
hence $(s,h) \models ls^0(x,y)$ **by** *auto*
thus $(s,\ h) \models Spat\ (ls(x,\ y))$ **by** *auto*
qed

lemma *one_entails_ls*: $[x{\neq}_p y] \mid [x\longmapsto y] \vdash Spat\ (ls(x,y))$
proof (*rule entailment_lift*)
define *xs* **where** *xs*: $xs = -\ (fv\ x \cup fv\ y)$
fix *s h*
assume *antecedent*: $(s,h) \models [x{\neq}_p y] \mid [x \longmapsto y]$
hence $[\![x]\!]s \neq [\![y]\!]s$ **by** *blast*
moreover from *antecedent* **obtain** v **where** $[\![x]\!]s = Val\ v\ h = [v \mapsto [\![y]\!]s]$ **by** *fastforce*
moreover have $h{=}h{+}{+}Map.empty\ h \perp Map.empty$ **by** *auto*
moreover have $\forall x' \in xs.\ (s(x'{:=}[\![y]\!]s),Map.empty) \models ls^0(\acute{}x'`,y)$
by (*metis EmptyLs dom_empty eval.simps*(1) *eval.simps*(2) *fun_upd_apply fv_expr.cases*)
moreover have $1 = Suc\ 0$ **by** *simp*
moreover from *xs* **have** $xs \subseteq -\ (fv\ x \cup fv\ y)$ **by** *simp*
ultimately have $(s,h) \models ls^1(x,y)$ **using** *ListSegment* **by** *blast*
thus $(s,h) \models Spat(ls(x,\ y))$ **by** *blast*
qed
end
theory *Proof_System*
imports *../basic_theory/Entailment*
begin

5 Rules of the Proof System

The proof system at the core of the decision procedure is based on the following rules.

theorem *axiom*: $\Pi|emp \vdash \top|emp$
by (*auto simp*: *entails_def*)

theorem *inconsistent*: $E{\neq}_p E \wedge_p \Pi|\Sigma \vdash F$
by (*auto simp*: *entails_def*)

theorem *substitution*: $subst\ x\ E\ (\Pi|\Sigma) \vdash subst\ x\ E\ (\Pi'|\Sigma') \Longrightarrow {}'x'{=}_p E \wedge_p \Pi|\Sigma \vdash \Pi'|\Sigma'$
proof (*rule entailment_lift*)
  fix $s\ h$
  assume *assm1*: $subst\ x\ E\ (\Pi|\Sigma) \vdash subst\ x\ E\ (\Pi'|\Sigma')$
  assume *assm2*: $(s,h) \models {}'x'{=}_p E \wedge_p \Pi|\Sigma$
  hence *s_eq*: $[\![{}'x']\!]s{=}[\![E]\!]s$ **and** $(s,h) \models \Pi|\Sigma$ **by** *fast+*
  hence $(s,h) \models subst\ x\ E\ (\Pi|\Sigma)$ **using** *subst_sat_rev* **by** *blast*
  with *assm1* **have** $(s,h) \models subst\ x\ E\ (\Pi'|\Sigma')$ **by** (*simp add*: *entails_def*)
  with *s_eq* **show** $(s,h) \models \Pi'|\Sigma'$ **using** *subst_sat* **by** *blast*
qed

theorem *eq_reflexivel*: $\Pi|\Sigma \vdash \Pi'|\Sigma' \Longrightarrow E{=}_p E \wedge_p \Pi|\Sigma \vdash \Pi'|\Sigma'$
unfolding *entails_def* **by** *blast*

theorem *eq_reflexiver*: $\Pi|\Sigma \vdash \Pi'|\Sigma' \Longrightarrow \Pi|\Sigma \vdash E{=}_p E \wedge_p \Pi'|\Sigma'$
unfolding *entails_def* **by** *blast*

theorem *hypothesis*: $P \wedge_p \Pi|\Sigma \vdash \Pi'|\Sigma' \Longrightarrow P \wedge_p \Pi|\Sigma \vdash P \wedge_p \Pi'|\Sigma'$
unfolding *entails_def* **by** *blast*

theorem *empty_ls*: $\Pi|\Sigma \vdash \Pi'|\Sigma' \Longrightarrow \Pi|\Sigma \vdash \Pi'|ls(E,E){*}\Sigma'$
unfolding *entails_def* **by** *fastforce*

theorem *nil_not_lval*: $E_1{\neq}_p nil \wedge_p \Pi|E_1{\longmapsto}E_2 * \Sigma \vdash \Pi'|\Sigma' \Longrightarrow \Pi|E_1{\longmapsto}E_2 * \Sigma \vdash \Pi'|\Sigma'$
unfolding *entails_def* **by** *force*

theorem *sep_conj_partial*: $E_1{\neq}_p E_3 \wedge_p \Pi|E_1{\longmapsto}E_2 * E_3{\longmapsto}E_4 * \Sigma \vdash \Pi'|\Sigma'$

  $\Longrightarrow \Pi|E_1{\longmapsto}E_2 * E_3{\longmapsto}E_4 * \Sigma \vdash \Pi'|\Sigma'$
proof (*rule entailment_lift*)
  fix $s\ h$
  assume *assm1*: $E_1{\neq}_p E_3 \wedge_p \Pi|E_1{\longmapsto}E_2 * E_3{\longmapsto}E_4 * \Sigma \vdash \Pi'|\Sigma'$
  assume *assm2*: $(s,h) \models \Pi|E_1{\longmapsto}E_2 * E_3{\longmapsto}E_4 * \Sigma$
   then obtain $h1\ h2$ **where** $h{=}h1{+}{+}h2\ \ h1{\perp}h2\ \ (s,h1) \models Spat(E_1{\longmapsto}E_2)$ $(s,h2) \models SpatF(E_3{\longmapsto}E_4 * \Sigma)$
    by *fastforce*
 moreover then obtain $h3\ h4$ **where** $h2{=}h3{+}{+}h4\ h3{\perp}h4\ (s,h3) \models Spat(E_3{\longmapsto}E_4)$
by *auto*

ultimately have $[\![E_1]\!]s \neq [\![E_3]\!]s$ **by** *fastforce*
hence $(s,h) \models Pure(E_1 \neq_p E_3)$ **by** *auto*
with *assm*2 **have** $(s,h) \models E_1 \neq_p E_3 \wedge_p \Pi | E_1 \longmapsto E_2 * E_3 \longmapsto E_4 * \Sigma$ **by** *auto*
with *assm*1 **show** $(s,h) \models \Pi' | \Sigma'$ **by** (*simp add*: *entails_def*)
qed

theorem *frame*: $\Pi|\Sigma \vdash \Pi'|\Sigma' \Longrightarrow \Pi|S * \Sigma \vdash \Pi'|S * \Sigma'$
proof (*rule entailment_lift*)
fix *s h*
assume *assm*1: $\Pi|\Sigma \vdash \Pi'|\Sigma'$
assume *assm*2: $(s,h) \models \Pi | S * \Sigma$
then obtain *h*1 *h*2 **where** *sep_conj*: $h = h1 ++ h2$ $h1 \perp h2$ $(s,h1) \models Spat\ S$ $(s,h2) \models SpatF\ \Sigma$ **by** *fast*
moreover {
from *assm*2 **have** $(s,h) \models PureF\ \Pi$ **by** *fastforce*
from *heap_puref*[*OF this*] **have** $(s,h2) \models PureF\ \Pi$ **by** *simp*
with *sep_conj*(4) **have** $(s,h2) \models \Pi|\Sigma$ **by** *fast*
with *assm*1 **have** $(s,h2) \models \Pi'|\Sigma'$ **by** (*simp add*: *entails_def*)
hence $(s,h2) \models PureF\ \Pi'$ $(s,h2) \models SpatF\ \Sigma'$ **by** *auto*
}
ultimately show $(s,h) \models \Pi'|S * \Sigma'$ **using** *heap_puref* **by** *blast*
qed

theorem *non_empty_ls*: $E_1 \neq_p E_3 \wedge_p \Pi|\Sigma \vdash \Pi'|ls(E_2,E_3) * \Sigma'$
$\Longrightarrow E_1 \neq_p E_3 \wedge_p \Pi|E_1 \longmapsto E_2 * \Sigma \vdash \Pi'|ls(E_1,E_3) * \Sigma'$
proof (*rule entailment_lift*)
define *xs* **where** *xs*: $xs = -(fv\ E_1 \cup fv\ E_2 \cup fv\ E_3)$
fix *s h*
assume *assm*1: $E_1 \neq_p E_3 \wedge_p \Pi|\Sigma \vdash \Pi'|ls(E_2,E_3) * \Sigma'$
assume *assm*2: $(s,h) \models E_1 \neq_p E_3 \wedge_p \Pi|E_1 \longmapsto E_2 * \Sigma$
then obtain *h*1 *h*2 **where** *sep_conj*:$h = h1 ++ h2$ $h1 \perp h2$ $(s,h1) \models Spat(E_1 \longmapsto E_2)$ $(s,h2) \models SpatF\ \Sigma$
by *fastforce*
then obtain *v* **where** *hd*: $[\![E_1]\!]s = Val\ v$ $h1 = [v \mapsto [\![E_2]\!]s]$ **by** *fast*
from *assm*2 **have** $[\![E_1]\!]s \neq [\![E_3]\!]s$ **by** *blast*
{
from *assm*2 **have** $(s,h) \models PureF\ (E_1 \neq_p E_3 \wedge_p \Pi)$ **by** *blast*
hence $(s,h2) \models PureF\ (E_1 \neq_p E_3 \wedge_p \Pi)$ **using** *heap_puref* **by** *auto*
with *sep_conj*(4) **have** $(s,h2) \models (E_1 \neq_p E_3 \wedge_p \Pi)|\Sigma$ **by** *auto*
with *assm*1 **have** $(s,h2) \models \Pi'|ls(E_2,E_3) * \Sigma'$ **by** (*simp add*: *entails_def*)
}
then obtain *h2_1 h2_2* **where** *sep_conj2*:$h2 = h2\_1 ++ h2\_2$ $h2\_1 \perp h2\_2$ $(s,h2\_1) \models Spat\ (ls(E_2,E_3))$
$(s,h2\_2) \models SpatF\ \Sigma'$ **by** *blast*

then obtain m **where** *tail*: $(s,h2\_1)\models ls^m(E_2,E_3)$ **by** *fast*
then obtain n **where** *n*: $n = Suc\ m$ **by** *simp*
have $\forall x \in xs.\ (s(x:=[\![E_2]\!]s),h2\_1)\models ls^m(\acute{}x`,E_3)$
proof
fix x
assume $x \in xs$
with *xs* **have** $x \notin fv\ E_2 \cup fv\ E_3$ **by** *fast*
from *ls_extend_rhs*[*OF tail this*] **have** *tail_ext*: $(s(x:=[\![E_2]\!]s),h2\_1)\models ls^m(E_2,E_3)$
.
have $[\![E_2]\!]s(x:=[\![E_2]\!]s) = [\![\acute{}x`]\!]s(x:=[\![E_2]\!]s)$
by (*metis eval.simps*(1) *eval.simps*(2) *expr.exhaust fun_upd_apply*)
from *ls_change_fst*[*OF tail_ext this*] **show** $(s(x:=[\![E_2]\!]s),h2\_1)\models ls^m(\acute{}x`,E_3)$
.
qed
from *sep_conj sep_conj2* **have** $h1\bot h2\_1$ **by** *auto*
from *xs* **have** $xs \subseteq -(fv\ E_1 \cup fv\ E_3)$ **by** *blast*
have $h1{+}{+}h2\_1=h1{+}{+}h2\_1$ **by** *simp*
from *ListSegment*[*OF hd* $\langle xs \subseteq -(fv\ E_1 \cup fv\ E_3)\rangle$ $\langle\forall x \in xs.\ (s(x:=[\![E_2]\!]s),h2\_1)\models ls^m(\acute{}x`,E_3)\rangle$ $\langle h1\bot h2\_1\rangle$
this n $\langle[\![E_1]\!]s\neq[\![E_3]\!]s\rangle$] **have** $(s,h1{+}{+}h2\_1)\models ls^n(E_1,E_3)$ **.**
{
hence $(s,h1{+}{+}h2\_1)\models Spat\ (ls(E_1,E_3))$ **by** *auto*
moreover from *sep_conj*(1,2) *sep_conj2*(1,2) **have** $h=h1{+}{+}h2\_1{+}{+}h2\_2$ $h1{+}{+}h2\_1\bot h2\_2$ **by** *auto*
ultimately have $(s,h)\models SpatF\ (ls(E_1,E_3) * \Sigma')$ **using** *sep_conj2*(4) **by** *blast*
}
moreover from $\langle(s,h2)\models\Pi'|ls(E_2,E_3) * \Sigma'\rangle$ **have** $(s,h)\models PureF\ \Pi'$ **using** *heap_puref* **by** *fast*
ultimately show $(s,h)\models\Pi'|ls(E_1,E_3) * \Sigma'$ **by** *blast*
qed

The most important rule UnrollCollapse:

theorem *UnrollCollapse*:
assumes $E_1{=_p}E_2 \wedge_p \Pi|\Sigma \vdash \Pi'|\Sigma'$
assumes $E_1{\neq_p}E_2 \wedge_p \acute{}x`{\neq_p}E_2 \wedge_p \Pi|E_1{\longmapsto}\acute{}x` * \acute{}x`{\longmapsto}E_2 * \Sigma \vdash \Pi'|\Sigma'$
assumes $x \notin fvl\ \Pi \cup fv\ E_1 \cup fv\ E_2 \cup fvl\ \Sigma \cup fvl\ \Pi' \cup fvl\ \Sigma'$
shows $\Pi|ls(E_1,E_2) * \Sigma \vdash \Pi'|\Sigma'$

sorry
end
theory *Sep_Log_Frag*
imports *decision_procedure/Proof_System*
begin

end
theory *Examples*
imports *../Sep_Log_Frag*
begin

6 Examples

In this section, some example entailments are either proven or disproven by usage of the decision procedures rules.

6.1 Auxiliary lemmata:

lemma *spatial_commut_entail*: $\Pi|(s1{*}s2{*}\Sigma) \vdash consequent \Longrightarrow \Pi|(s2{*}s1{*}\Sigma) \vdash consequent$
using *entails_def spatial_commut_form* **by** *force*
lemma *pure_commut_entail*: $(p1{\wedge_p}p2{\wedge_p}\Pi)|\Sigma \vdash consequent \Longrightarrow (p2{\wedge_p}p1{\wedge_p}\Pi)|\Sigma \vdash consequent$
using *entails_def pure_commut_form* **by** *simp*

6.2 First example (cf. [3] 2.4)

lemma $[\acute{}\,''x''\text{`}{\neq_p}\acute{}\,p\text{`}]|\acute{}\,''x''\text{`}{\longmapsto}\acute{}\,''y''\text{`} * \acute{}\,''y''\text{`}{\longmapsto}nil * emp \vdash [\,]|[ls(\acute{}\,''x''\text{`},nil)]$
— First enrich the formula to contain all necessary inequalities, ...
apply (*rule nil_not_lval*)
apply (*rule spatial_commut_entail*)
apply (*rule nil_not_lval*)
— ... then destructure the ls step by step, ...
apply (*rule pure_commut_entail*)
apply (*rule spatial_commut_entail*)
apply (*rule non_empty_ls*)
apply (*rule pure_commut_entail*)
apply (*rule non_empty_ls*)
— ... then exchange the empty ls with emp ...
apply (*rule empty_ls*)
— ... and finally prove the entailment with the axiom.
by (*rule axiom*)

6.3 Second example (cf. [3] 2.4)

At first try to prove this by applying the decision procedure as far as possible.

lemma $(\top|\acute{}''x''`\longmapsto nil * \acute{}''y''`\longmapsto nil * emp \vdash [\acute{}''x''`=_p\acute{}''y''`]|[\acute{}''y''`\longmapsto nil])$
— First enrich the formula to contain all necessary inequalities, ...
apply (*rule sep_conj_partial*)
apply (*rule nil_not_lval*)
apply (*rule spatial_commut_entail*)
apply (*rule nil_not_lval*)
— ... then remove the duplicate $y \mapsto nil$ ...
apply (*rule frame*)
— ... and there is no applicable rule and so the decision procedure halts.
sorry

The resulting transformed goal can no be proven wrong with a counter example.

lemma $\neg([\acute{}''y''`\neq_p nil, \acute{}''x''`\neq_p nil, \acute{}''x''`\neq_p\acute{}''y''`]|[\acute{}''x''`\longmapsto nil] \vdash [\acute{}''x''`=_p\acute{}''y''`]|emp)$

  (**is** $\neg(?ant \vdash ?cons)$)
proof
  assume $?ant \vdash ?cons$
  hence *hyp*: $\forall s\ h.\ (s,h)\models ?ant \longrightarrow (s,h)\models ?cons$ **using** *entails_def* **by** *simp*
  define *s h* **where** *s_def*:$(s::stack) = ((\lambda\_.\ Nilval)(''x'':= Val\ 5))(''y'':= Val\ 23)$
    and *h_def*: $(h::heap) = [5\mapsto Nilval]$
  have $(s,h)\models ?ant$ **proof** — The state (s,h) satisfies the antecedent ...
    show $(s,h)\models PureF\ (\acute{}''y''`\neq_p nil \wedge_p \acute{}''x''`\neq_p nil \wedge_p \acute{}''x''`\neq_p\acute{}''y''` \wedge_p \top)$ **proof**
      show $(s,h)\models Pure\ (\acute{}''y''`\neq_p nil)$ **by** (*simp add*: *NeqSat s_def*)
    next
      show $(s,h)\models PureF\ (\acute{}''x''`\neq_p nil \wedge_p \acute{}''x''`\neq_p\acute{}''y''` \wedge_p \top)$ **proof**
        show $(s,h)\models Pure\ (\acute{}''x''`\neq_p nil)$ **by** (*simp add*: *NeqSat s_def*)
      next
        show $(s,h)\models PureF\ (\acute{}''x''`\neq_p\acute{}''y''` \wedge_p \top)$ **proof**
          show $(s,h)\models Pure\ (\acute{}''x''`\neq_p\acute{}''y''`)$ **by** (*simp add*: *NeqSat s_def*)
        next
          show $(s,h)\models PureF\ (\top)$ **by** *auto*
        qed
      qed
    qed
  next
    define *h1 h2* **where** *h_defs*: $h1 = h\ (h2::heap) = Map.empty$
    hence $h1\bot h2\ h=h1{+}{+}h2$ **by** *simp_all*
    moreover from *h_defs* **have** $(s,h2)\models SpatF\ emp$ **by** *auto*
    moreover have $(s,h1)\models Spat\ (\acute{}''x''`\longmapsto nil)$ **proof**
      show $[\![\acute{}''x''`]\!]s = Val\ 5$ **using** *s_def* **by** *simp*
    next

show $h1 = [5 \mapsto [\![nil]\!]s]$ **using** *h_defs h_def* **by** *simp*
qed
ultimately show $(s,h)\models SpatF\ ('''x''' \longmapsto nil * emp)$ **by** *blast*
qed
with *hyp* **have** $(s,h)\models ?cons$ **by** *simp*
moreover have *contradiction*: $\neg(s,h)\models ?cons$ **proof** — ... but not the consequent.
assume $(s,h)\models ?cons$
hence $(s,h)\models Pure\ ('''x''' =_p '''y''')$ **by** *auto*
hence $[\!['''x''']\!]s = [\!['''y''']\!]s$ **by** *auto*
moreover from *s_def* **have** $[\!['''x''']\!]s \neq [\!['''y''']\!]s$ **by** *simp*
ultimately show *False* **by** *simp*
qed
ultimately show *False* **by** *simp*
qed
end
theory *Model_Theory*
imports *../basic_theory/Entailment Proof_System*
begin

7 Decidability, Model-Theoretical

This section contains proofs from the corresponding section 3 in [1] about decidability, soundness and completeness.

Entailments without ls in the antecedent are decidable, cf. Lemma 3 [1].

theorem *entailment_decidable_simple*: $\bigwedge \Pi\ \Sigma\ \Pi'\ \Sigma'.\ \nexists E_1\ E_2.\ ls(E_1,E_2) \in set\ \Sigma$
$\Longrightarrow \Pi|\Sigma \vdash \Pi'|\Sigma' \vee \neg(\Pi|\Sigma \vdash \Pi'|\Sigma')$
unfolding *entails_def* **by** *blast*

7.1 Proof of the above theorem as in [1], A.2.

definition *state_equiv* :: $state \Rightarrow var\ set \Rightarrow state \Rightarrow bool$ **where**
$state\_equiv\ sh\ X\ sh' \equiv \exists\, s\ h\ s'\ h'.\ sh = (s,h) \wedge sh' = (s',h') \wedge (\exists\, r.\ bij\ r \wedge r\ Nilval = Nilval$
$\wedge (\forall x \in X.\ r\ (s\ x) = s'\ x) \wedge (\forall\, l\ l'.\ (l \notin dom\ h \wedge r\ (Val\ l) = Val\ l' \wedge l' \notin dom\ h')$
$\vee ((r\ (Val\ l))) = Val\ l' \wedge r\ (the\ (h\ l)) = the\ (h'\ l')))$
notation *state_equiv* (_ $\approx_{\_}$ _ [60,61,61] 61)

cf Lemma 19 in [1], A.2

lemma $[\![X=fv\ A;\ (s,h)\approx_X(s',h')]\!] \Longrightarrow ((s,h)\models A) = ((s',h')\models A)$
proof

assume X: X=*fv* A
assume *equiv*: $(s,h) \approx_X (s',h')$
assume *lhs*: $(s,h) \models A$
from *lhs equiv* X **show** $(s',h') \models A$ **proof** (*induction rule*: *sat_induct*)
case (*EqSat* *e*1 *s* *e*2 *h*)
then obtain r **where** r: (*bij* r $\wedge$ r *Nilval* = *Nilval*
$\wedge$ ($\forall x \in X.$ r (s x) = s' x) $\wedge$ ($\forall l\ l'.$ ($l \notin$ *dom* h $\wedge$ r (*Val* l) = *Val* l' $\wedge$ $l' \notin$ *dom* h')
$\vee$ ((r (*Val* l))) = *Val* l' $\wedge$ r (*the* (h l)) = *the* (h' l'))) **using** *state_equiv_def* **by** *auto*
with *EqSat.prems*(2) **have** *r_eq*: $\forall x \in$ *fv* *e*1 $\cup$ *fv* *e*2. r (s x) = s' x **by** *simp*
show *?case* **proof** (*cases* *e*1)
case *Nil*
hence *e*1*_nil*: $[\![e1]\!]s'$ = *Nilval* **by** *simp*
from *Nil* **have** *s_e*1*_nil*: $[\![e1]\!]s$ = *Nilval* **by** *simp*
then show *?thesis* **proof** (*cases* *e*2)
case *Nil*
hence $[\![e2]\!]s'$ = *Nilval* **by** *simp*
with *e*1*_nil* **show** *?thesis* **by** *fastforce*
next
case (*Var* *x*2)
with *s_e*1*_nil* *EqSat.hyps* **have** s *x*2 = *Nilval* **by** *simp*
moreover with *r_eq* **have** r (s *x*2) = s' *x*2 **by** (*simp add*: *Var*)
ultimately have $[\![e2]\!]s'$ = *Nilval* **using** r *Var* **by** *simp*
with *e*1*_nil* **show** *?thesis* **by** *fastforce*
qed
next
case (*Var* a)
with *r_eq* **have** r (s a) = s' a **by** *simp*
hence *e*1*_eq*: $[\![e1]\!]s' = r$ ($[\![e1]\!]s$) **using** *Var* **by** *simp*
then show *?thesis* **proof** (*cases* *e*2)
case *Nil*
hence $[\![e2]\!]s$ = *Nilval* **by** *simp*
with *EqSat.hyps* **have** $[\![e1]\!]s$ = *Nilval* **by** *simp*
with *e*1*_eq* r **have** $[\![e1]\!]s'$ = *Nilval* **by** *simp*
moreover from *Nil* **have** $[\![e2]\!]s'$ = *Nilval* **by** *fastforce*
ultimately show *?thesis* **by** *fastforce*
next
case (*Var* b)
with *r_eq* **have** r (s b) = s' b **by** *simp*
hence $[\![e2]\!]s' = r$ ($[\![e2]\!]s$) **using** *Var* **by** *simp*
with *e*1*_eq* *EqSat.hyps* **show** *?thesis* **by** *auto*
qed

qed
next
case (*NeqSat* $e1$ s $e2$ h)
then show *?case* **sorry**
next
case (*TrueSat* s h)
then show *?case* **sorry**
next
case (*ConjSat* s h P Π)
then show *?case* **sorry**
next
case (*PointsToSat* $e1$ s v h $e2$)
then show *?case* **sorry**
next
case (*EmpSat* h s)
then show *?case* **sorry**
next
case (*SepConjSat* $h1$ $h2$ h s S Σ)
then show *?case* **sorry**
next
case (*FormSat* s h Π Σ)
then show *?case* **sorry**
next
case (*LsSat* s h n $e1$ $e2$)
then show *?case* **sorry**
qed
next
assume X: X=*fv* A
assume *equiv*: $(s,h) \approx_X (s',h')$
assume *rhs*: $(s',h') \models A$
show $(s,h) \models A$ — Works pretty much the same as above, but one has to reverse r.
sorry
qed
end

References

[1] Berdine, J., Calcagno, C., and O'Hearn, P. W. A Decidable Fragment of Separation Logic. In *Foundations of Software Technology and Theoretical Computer Science* (Berlin, Heidelberg, 2004), K. Lodaya and M. Mahajan, Eds., vol. 3328 of *Lecture Notes in Computer Science*, Springer, pp. 97–109.

[2] Sextl, F. Alice_rs github repository. https://github.com/firefighterduck/alice_rs.

[3] Sextl, F. A Decidable Fragment of Separation Logic. https://www21.in.tum.de/teaching/sar/SS20/8.pdf, 2020. Seminar Automated Reasoning.