Separation Algebra

Gerwin Klein and Rafal Kolanski and Andrew Boyton

March 19, 2025

Abstract

We present a generic type class implementation of separation algebra for Isabelle/HOL as well as lemmas and generic tactics which can be used directly for any instantiation of the type class.

The ex directory contains example instantiations that include structures such as a heap or virtual memory.

The abstract separation algebra is based upon "Abstract Separation Logic" by Calcagno et al. These theories are also the basis of "Mechanised Separation Algebra" by the authors [1].

The aim of this work is to support and significantly reduce the effort for future separation logic developments in Isabelle/HOL by factoring out the part of separation logic that can be treated abstractly once and for all. This includes developing typical default rule sets for reasoning as well as automated tactic support for separation logic.

Contents

1 Abstract Separation Algebra

theory *Separation-Algebra*
imports *Main*
begin

This theory is the main abstract separation algebra development

2 Input syntax for lifting boolean predicates to separation predicates

abbreviation (*input*)
pred-and :: ($'a \Rightarrow bool$) $\Rightarrow$ ($'a \Rightarrow bool$) $\Rightarrow$ $'a \Rightarrow bool$ (**infixr** ‹*and*› *35*) **where**
a *and* $b \equiv \lambda s.\ a\ s \wedge b\ s$

abbreviation (*input*)
pred-or :: ($'a \Rightarrow bool$) $\Rightarrow$ ($'a \Rightarrow bool$) $\Rightarrow$ $'a \Rightarrow bool$ (**infixr** ‹*or*› *30*) **where**
a *or* $b \equiv \lambda s.\ a\ s \vee b\ s$

abbreviation (*input*)
pred-not :: ($'a \Rightarrow bool$) $\Rightarrow$ $'a \Rightarrow bool$ (‹*not* -› [*40*] *40*) **where**
not $a \equiv \lambda s.\ \neg a\ s$

abbreviation (*input*)
pred-imp :: ($'a \Rightarrow bool$) $\Rightarrow$ ($'a \Rightarrow bool$) $\Rightarrow$ $'a \Rightarrow bool$ (**infixr** ‹*imp*› *25*) **where**
a *imp* $b \equiv \lambda s.\ a\ s \longrightarrow b\ s$

abbreviation (*input*)

pred-K :: ′*b* ⇒ ′*a* ⇒ ′*b* (‹⟨-⟩›) **where**
⟨*f*⟩ ≡ λ*s*. *f*

abbreviation (*input*)
pred-ex :: (′*b* ⇒ ′*a* ⇒ *bool*) ⇒ ′*a* ⇒ *bool* (**binder** ‹*EXS* › *10*) **where**
EXS x. *P x* ≡ λ*s*. ∃ *x*. *P x s*

abbreviation (*input*)
pred-all :: (′*b* ⇒ ′*a* ⇒ *bool*) ⇒ ′*a* ⇒ *bool* (**binder** ‹*ALLS* › *10*) **where**
ALLS x. *P x* ≡ λ*s*. ∀ *x*. *P x s*

3 Associative/Commutative Monoid Basis of Separation Algebras

class *pre-sep-algebra* = *zero* + *plus* +
fixes *sep-disj* :: ′*a* => ′*a* => *bool* (**infix** ‹##› *60*)

assumes *sep-disj-zero* [*simp*]: *x* ## *0*
assumes *sep-disj-commuteI*: *x* ## *y* ⟹ *y* ## *x*

assumes *sep-add-zero* [*simp*]: *x* + *0* = *x*
assumes *sep-add-commute*: *x* ## *y* ⟹ *x* + *y* = *y* + *x*

assumes *sep-add-assoc*:
⟦ *x* ## *y*; *y* ## *z*; *x* ## *z* ⟧ ⟹ (*x* + *y*) + *z* = *x* + (*y* + *z*)
begin

lemma *sep-disj-commute*: *x* ## *y* = *y* ## *x*
⟨*proof*⟩

lemma *sep-add-left-commute*:
assumes *a*: *a* ## *b b* ## *c a* ## *c*
shows *b* + (*a* + *c*) = *a* + (*b* + *c*) (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemmas *sep-add-ac* = *sep-add-assoc sep-add-commute sep-add-left-commute*
sep-disj-commute

end

4 Separation Algebra as Defined by Calcagno et al.

class *sep-algebra* = *pre-sep-algebra* +
assumes *sep-disj-addD1*: ⟦ *x* ## *y* + *z*; *y* ## *z* ⟧ ⟹ *x* ## *y*
assumes *sep-disj-addI1*: ⟦ *x* ## *y* + *z*; *y* ## *z* ⟧ ⟹ *x* + *y* ## *z*
begin

4.1 Basic Construct Definitions and Abbreviations

definition
sep-conj :: $('a \Rightarrow bool) \Rightarrow ('a \Rightarrow bool) \Rightarrow ('a \Rightarrow bool)$ (**infixr** ‹**› *35*)
where
$P ** Q \equiv \lambda h.\ \exists x\ y.\ x \mathbin{\#\#} y \wedge h = x + y \wedge P\ x \wedge Q\ y$

notation
sep-conj (**infixr** ‹∧*› *35*)

definition
sep-empty :: $'a \Rightarrow bool$ (‹□›) **where**
$\square \equiv \lambda h.\ h = 0$

definition
sep-impl :: $('a \Rightarrow bool) \Rightarrow ('a \Rightarrow bool) \Rightarrow ('a \Rightarrow bool)$ (**infixr** ‹⟶*› *25*)
where
$P \longrightarrow* Q \equiv \lambda h.\ \forall h'.\ h \mathbin{\#\#} h' \wedge P\ h' \longrightarrow Q\ (h + h')$

definition
sep-substate :: $'a => 'a => bool$ (**infix** ‹⪯› *60*) **where**
$x \preceq y \equiv \exists z.\ x \mathbin{\#\#} z \wedge x + z = y$

abbreviation
sep-true ≡ ⟨*True*⟩

abbreviation
sep-false ≡ ⟨*False*⟩

definition
sep-list-conj :: $('a \Rightarrow bool)\ list \Rightarrow ('a \Rightarrow bool)$ (‹⋀* -› [*60*] *90*) **where**
sep-list-conj Ps ≡ *foldl* (**) □ *Ps*

4.2 Disjunction/Addition Properties

lemma *disjoint-zero-sym* [*simp*]: $0 \mathbin{\#\#} x$
⟨*proof*⟩

lemma *sep-add-zero-sym* [*simp*]: $0 + x = x$
⟨*proof*⟩

lemma *sep-disj-addD2*: $[\![x \mathbin{\#\#} y + z; y \mathbin{\#\#} z]\!] \Longrightarrow x \mathbin{\#\#} z$
⟨*proof*⟩

lemma *sep-disj-addD*: $[\![x \mathbin{\#\#} y + z; y \mathbin{\#\#} z]\!] \Longrightarrow x \mathbin{\#\#} y \wedge x \mathbin{\#\#} z$
⟨*proof*⟩

lemma *sep-add-disjD*: $[\![x + y \mathbin{\#\#} z; x \mathbin{\#\#} y]\!] \Longrightarrow x \mathbin{\#\#} z \wedge y \mathbin{\#\#} z$
⟨*proof*⟩

lemma *sep-disj-addI2*:
$\llbracket\ x\ \#\#\ y + z;\ y\ \#\#\ z\ \rrbracket \Longrightarrow x + z\ \#\#\ y$
⟨*proof*⟩

lemma *sep-add-disjI1*:
$\llbracket\ x + y\ \#\#\ z;\ x\ \#\#\ y\ \rrbracket \Longrightarrow x + z\ \#\#\ y$
⟨*proof*⟩

lemma *sep-add-disjI2*:
$\llbracket\ x + y\ \#\#\ z;\ x\ \#\#\ y\ \rrbracket \Longrightarrow z + y\ \#\#\ x$
⟨*proof*⟩

lemma *sep-disj-addI3*:
$x + y\ \#\#\ z \Longrightarrow x\ \#\#\ y \Longrightarrow x\ \#\#\ y + z$
⟨*proof*⟩

lemma *sep-disj-add*:
$\llbracket\ y\ \#\#\ z;\ x\ \#\#\ y\ \rrbracket \Longrightarrow x\ \#\#\ y + z = x + y\ \#\#\ z$
⟨*proof*⟩

4.3 Substate Properties

lemma *sep-substate-disj-add*:
$x\ \#\#\ y \Longrightarrow x \preceq x + y$
⟨*proof*⟩

lemma *sep-substate-disj-add'*:
$x\ \#\#\ y \Longrightarrow x \preceq y + x$
⟨*proof*⟩

4.4 Separating Conjunction Properties

lemma *sep-conjD*:
$(P \wedge\!* \ Q)\ h \Longrightarrow \exists x\ y.\ x\ \#\#\ y \wedge h = x + y \wedge P\ x \wedge Q\ y$
⟨*proof*⟩

lemma *sep-conjE*:
$\llbracket\ (P ** Q)\ h;\ \bigwedge x\ y.\ \llbracket\ P\ x;\ Q\ y;\ x\ \#\#\ y;\ h = x + y\ \rrbracket \Longrightarrow X\ \rrbracket \Longrightarrow X$
⟨*proof*⟩

lemma *sep-conjI*:
$\llbracket\ P\ x;\ Q\ y;\ x\ \#\#\ y;\ h = x + y\ \rrbracket \Longrightarrow (P ** Q)\ h$
⟨*proof*⟩

lemma *sep-conj-commuteI*:
$(P ** Q)\ h \Longrightarrow (Q ** P)\ h$
⟨*proof*⟩

lemma *sep-conj-commute*:

$(P ** Q) = (Q ** P)$
⟨*proof*⟩

lemma *sep-conj-assoc*:
$((P ** Q) ** R) = (P ** Q ** R)$ (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *sep-conj-impl*:
$[\![(P ** Q)\ h;\ \bigwedge h.\ P\ h \Longrightarrow P'\ h;\ \bigwedge h.\ Q\ h \Longrightarrow Q'\ h]\!] \Longrightarrow (P' ** Q')\ h$
⟨*proof*⟩

lemma *sep-conj-impl1*:
assumes P: $\bigwedge h.\ P\ h \Longrightarrow I\ h$
shows $(P ** R)\ h \Longrightarrow (I ** R)\ h$
⟨*proof*⟩

lemma *sep-globalise*:
$[\![(P ** R)\ h;\ (\bigwedge h.\ P\ h \Longrightarrow Q\ h)]\!] \Longrightarrow (Q ** R)\ h$
⟨*proof*⟩

lemma *sep-conj-trivial-strip2*:
$Q = R \Longrightarrow (Q ** P) = (R ** P)$ ⟨*proof*⟩

lemma *disjoint-subheaps-exist*:
$\exists\, x\ y.\ x\ \#\#\ y \wedge h = x + y$
⟨*proof*⟩

lemma *sep-conj-left-commute*:
$(P ** (Q ** R)) = (Q ** (P ** R))$ (**is** *?x* = *?y*)
⟨*proof*⟩

lemmas *sep-conj-ac* = *sep-conj-commute sep-conj-assoc sep-conj-left-commute*

lemma *ab-semigroup-mult-sep-conj*: *class.ab-semigroup-mult* $(**)$
⟨*proof*⟩

lemma *sep-empty-zero* [*simp*,*intro*!]: □ *0*
⟨*proof*⟩

4.5 Properties of *sep-true* and *sep-false*

lemma *sep-conj-sep-true*:
$P\ h \Longrightarrow (P ** \textit{sep-true})\ h$
⟨*proof*⟩

lemma *sep-conj-sep-true'*:
$P\ h \Longrightarrow (\textit{sep-true} ** P)\ h$
⟨*proof*⟩

lemma *sep-conj-true* [*simp*]:
$(sep\text{-}true ** sep\text{-}true) = sep\text{-}true$
⟨*proof*⟩

lemma *sep-conj-false-right* [*simp*]:
$(P ** sep\text{-}false) = sep\text{-}false$
⟨*proof*⟩

lemma *sep-conj-false-left* [*simp*]:
$(sep\text{-}false ** P) = sep\text{-}false$
⟨*proof*⟩

4.6 Properties of zero (□)

lemma *sep-conj-empty* [*simp*]:
$(P ** \square) = P$
⟨*proof*⟩

lemma *sep-conj-empty'*[*simp*]:
$(\square ** P) = P$
⟨*proof*⟩

lemma *sep-conj-sep-emptyI*:
$P\ h \Longrightarrow (P ** \square)\ h$
⟨*proof*⟩

lemma *sep-conj-sep-emptyE*:
$[\![\ P\ s;\ (P ** \square)\ s \Longrightarrow (Q ** R)\ s\ ]\!] \Longrightarrow (Q ** R)\ s$
⟨*proof*⟩

lemma *monoid-add*: *class.monoid-add* $((**))\ \square$
⟨*proof*⟩

lemma *comm-monoid-add*: *class.comm-monoid-add* $(**)\ \square$
⟨*proof*⟩

4.7 Properties of top (*sep-true*)

lemma *sep-conj-true-P* [*simp*]:
$(sep\text{-}true ** (sep\text{-}true ** P)) = (sep\text{-}true ** P)$
⟨*proof*⟩

lemma *sep-conj-disj*:
$((P\ or\ Q) ** R) = ((P ** R)\ or\ (Q ** R))$
⟨*proof*⟩

lemma *sep-conj-sep-true-left*:
$(P ** Q)\ h \Longrightarrow (sep\text{-}true ** Q)\ h$
⟨*proof*⟩

lemma *sep-conj-sep-true-right*:
$(P ** Q)\ h \Longrightarrow (P ** \mathit{sep\text{-}true})\ h$
⟨*proof*⟩

4.8 Separating Conjunction with Quantifiers

lemma *sep-conj-conj*:
$((P\ \mathit{and}\ Q) ** R)\ h \Longrightarrow ((P ** R)\ \mathit{and}\ (Q ** R))\ h$
⟨*proof*⟩

lemma *sep-conj-exists1*:
$((\mathit{EXS}\ x.\ P\ x) ** Q) = (\mathit{EXS}\ x.\ (P\ x ** Q))$
⟨*proof*⟩

lemma *sep-conj-exists2*:
$(P ** (\mathit{EXS}\ x.\ Q\ x)) = (\mathit{EXS}\ x.\ P ** Q\ x)$
⟨*proof*⟩

lemmas *sep-conj-exists* = *sep-conj-exists1 sep-conj-exists2*

lemma *sep-conj-spec*:
$((\mathit{ALLS}\ x.\ P\ x) ** Q)\ h \Longrightarrow (P\ x ** Q)\ h$
⟨*proof*⟩

4.9 Properties of Separating Implication

lemma *sep-implI*:
assumes *a*: $\bigwedge h'.\ [\![\ h\ \#\#\ h';\ P\ h'\ ]\!] \Longrightarrow Q\ (h + h')$
shows $(P \longrightarrow* Q)\ h$
⟨*proof*⟩

lemma *sep-implD*:
$(x \longrightarrow* y)\ h \Longrightarrow \forall h'.\ h\ \#\#\ h' \wedge x\ h' \longrightarrow y\ (h + h')$
⟨*proof*⟩

lemma *sep-implE*:
$(x \longrightarrow* y)\ h \Longrightarrow (\forall h'.\ h\ \#\#\ h' \wedge x\ h' \longrightarrow y\ (h + h') \Longrightarrow Q) \Longrightarrow Q$
⟨*proof*⟩

lemma *sep-impl-sep-true* [*simp*]:
$(P \longrightarrow* \mathit{sep\text{-}true}) = \mathit{sep\text{-}true}$
⟨*proof*⟩

lemma *sep-impl-sep-false* [*simp*]:
$(\mathit{sep\text{-}false} \longrightarrow* P) = \mathit{sep\text{-}true}$
⟨*proof*⟩

lemma *sep-impl-sep-true-P*:
$(\mathit{sep\text{-}true} \longrightarrow* P)\ h \Longrightarrow P\ h$
⟨*proof*⟩

lemma *sep-impl-sep-true-false* [*simp*]:
$(\textit{sep-true} \longrightarrow\!* \textit{ sep-false}) = \textit{sep-false}$
⟨*proof*⟩

lemma *sep-conj-sep-impl*:
$[\![P\ h;\ \bigwedge h.\ (P\ {**}\ Q)\ h \Longrightarrow R\ h\ ]\!] \Longrightarrow (Q \longrightarrow\!*\ R)\ h$
⟨*proof*⟩

lemma *sep-conj-sep-impl2*:
$[\![(P\ {**}\ Q)\ h;\ \bigwedge h.\ P\ h \Longrightarrow (Q \longrightarrow\!*\ R)\ h\ ]\!] \Longrightarrow R\ h$
⟨*proof*⟩

lemma *sep-conj-sep-impl-sep-conj2*:
$(P\ {**}\ R)\ h \Longrightarrow (P\ {**}\ (Q \longrightarrow\!*\ (Q\ {**}\ R)))\ h$
⟨*proof*⟩

4.10 Pure assertions

definition
$\textit{pure} :: ('a \Rightarrow \textit{bool}) \Rightarrow \textit{bool}$ **where**
$\textit{pure}\ P \equiv \forall\, h\ h'.\ P\ h = P\ h'$

lemma *pure-sep-true*:
pure sep-true
⟨*proof*⟩

lemma *pure-sep-false*:
pure sep-true
⟨*proof*⟩

lemma *pure-split*:
$\textit{pure}\ P = (P = \textit{sep-true} \lor P = \textit{sep-false})$
⟨*proof*⟩

lemma *pure-sep-conj*:
$[\![\textit{pure}\ P;\ \textit{pure}\ Q\ ]\!] \Longrightarrow \textit{pure}\ (P \land\!*\ Q)$
⟨*proof*⟩

lemma *pure-sep-impl*:
$[\![\textit{pure}\ P;\ \textit{pure}\ Q\ ]\!] \Longrightarrow \textit{pure}\ (P \longrightarrow\!*\ Q)$
⟨*proof*⟩

lemma *pure-conj-sep-conj*:
$[\![(P\ \textit{and}\ Q)\ h;\ \textit{pure}\ P \lor \textit{pure}\ Q\ ]\!] \Longrightarrow (P \land\!*\ Q)\ h$
⟨*proof*⟩

lemma *pure-sep-conj-conj*:
$[\![(P \land\!*\ Q)\ h;\ \textit{pure}\ P;\ \textit{pure}\ Q\ ]\!] \Longrightarrow (P\ \textit{and}\ Q)\ h$

$\langle proof \rangle$

lemma *pure-conj-sep-conj-assoc*:
$pure\ P \Longrightarrow ((P\ and\ Q) \wedge\!*\ R) = (P\ and\ (Q \wedge\!*\ R))$
$\langle proof \rangle$

lemma *pure-sep-impl-impl*:
$[\![\ (P \longrightarrow\!*\ Q)\ h;\ pure\ P\ ]\!] \Longrightarrow P\ h \longrightarrow Q\ h$
$\langle proof \rangle$

lemma *pure-impl-sep-impl*:
$[\![\ P\ h \longrightarrow Q\ h;\ pure\ P;\ pure\ Q\ ]\!] \Longrightarrow (P \longrightarrow\!*\ Q)\ h$
$\langle proof \rangle$

lemma *pure-conj-right*: $(Q \wedge\!*\ (\langle P' \rangle\ and\ Q')) = (\langle P' \rangle\ and\ (Q \wedge\!*\ Q'))$
$\langle proof \rangle$

lemma *pure-conj-right'*: $(Q \wedge\!*\ (P'\ and\ \langle Q' \rangle)) = (\langle Q' \rangle\ and\ (Q \wedge\!*\ P'))$
$\langle proof \rangle$

lemma *pure-conj-left*: $((\langle P' \rangle\ and\ Q') \wedge\!*\ Q) = (\langle P' \rangle\ and\ (Q' \wedge\!*\ Q))$
$\langle proof \rangle$

lemma *pure-conj-left'*: $((P'\ and\ \langle Q' \rangle) \wedge\!*\ Q) = (\langle Q' \rangle\ and\ (P' \wedge\!*\ Q))$
$\langle proof \rangle$

lemmas *pure-conj* = *pure-conj-right pure-conj-right' pure-conj-left pure-conj-left'*

declare *pure-conj*[*simp add*]

4.11 Intuitionistic assertions

definition *intuitionistic* :: $('a \Rightarrow bool) \Rightarrow bool$ **where**
$intuitionistic\ P \equiv \forall\, h\ h'.\ P\ h \wedge h \preceq h' \longrightarrow P\ h'$

lemma *intuitionisticI*:
$(\bigwedge h\ h'.\ [\![\ P\ h;\ h \preceq h'\ ]\!] \Longrightarrow P\ h') \Longrightarrow intuitionistic\ P$
$\langle proof \rangle$

lemma *intuitionisticD*:
$[\![\ intuitionistic\ P;\ P\ h;\ h \preceq h'\ ]\!] \Longrightarrow P\ h'$
$\langle proof \rangle$

lemma *pure-intuitionistic*:
$pure\ P \Longrightarrow intuitionistic\ P$
$\langle proof \rangle$

lemma *intuitionistic-conj*:

$\llbracket$ *intuitionistic P*; *intuitionistic Q* $\rrbracket \Longrightarrow$ *intuitionistic* (*P and Q*)
⟨*proof*⟩

lemma *intuitionistic-disj*:
$\llbracket$ *intuitionistic P*; *intuitionistic Q* $\rrbracket \Longrightarrow$ *intuitionistic* (*P or Q*)
⟨*proof*⟩

lemma *intuitionistic-forall*:
($\bigwedge x$. *intuitionistic* (*P x*)) $\Longrightarrow$ *intuitionistic* (*ALLS x. P x*)
⟨*proof*⟩

lemma *intuitionistic-exists*:
($\bigwedge x$. *intuitionistic* (*P x*)) $\Longrightarrow$ *intuitionistic* (*EXS x. P x*)
⟨*proof*⟩

lemma *intuitionistic-sep-conj-sep-true*:
intuitionistic (*sep-true* ∧∗ *P*)
⟨*proof*⟩

lemma *intuitionistic-sep-impl-sep-true*:
intuitionistic (*sep-true* ⟶∗ *P*)
⟨*proof*⟩

lemma *intuitionistic-sep-conj*:
assumes *ip*: *intuitionistic* (*P*::(*'a* ⇒ *bool*))
shows *intuitionistic* (*P* ∧∗ *Q*)
⟨*proof*⟩

lemma *intuitionistic-sep-impl*:
assumes *iq*: *intuitionistic Q*
shows *intuitionistic* (*P* ⟶∗ *Q*)
⟨*proof*⟩

lemma *strongest-intuitionistic*:
¬ (∃ *Q*. (∀ *h*. (*Q h* ⟶ (*P* ∧∗ *sep-true*) *h*)) ∧ *intuitionistic Q* ∧
Q ≠ (*P* ∧∗ *sep-true*) ∧ (∀ *h*. *P h* ⟶ *Q h*))
⟨*proof*⟩

lemma *weakest-intuitionistic*:
¬ (∃ *Q*. (∀ *h*. ((*sep-true* ⟶∗ *P*) *h* ⟶ *Q h*)) ∧ *intuitionistic Q* ∧
Q ≠ (*sep-true* ⟶∗ *P*) ∧ (∀ *h*. *Q h* ⟶ *P h*))
⟨*proof*⟩

lemma *intuitionistic-sep-conj-sep-true-P*:
$\llbracket$ (*P* ∧∗ *sep-true*) *s*; *intuitionistic P* $\rrbracket \Longrightarrow$ *P s*
⟨*proof*⟩

lemma *intuitionistic-sep-conj-sep-true-simp*:
intuitionistic P $\Longrightarrow$ (*P* ∧∗ *sep-true*) = *P*

⟨*proof*⟩

lemma *intuitionistic-sep-impl-sep-true-P*:
$\llbracket$ $P\ h$; *intuitionistic* P $\rrbracket \Longrightarrow$ (*sep-true* $\longrightarrow*$ P) h
⟨*proof*⟩

lemma *intuitionistic-sep-impl-sep-true-simp*:
intuitionistic $P \Longrightarrow$ (*sep-true* $\longrightarrow*$ P) $= P$
⟨*proof*⟩

4.12 Strictly exact assertions

definition *strictly-exact* :: ($'a \Rightarrow$ *bool*) $\Rightarrow$ *bool* **where**
strictly-exact $P \equiv \forall h\ h'.\ P\ h \wedge P\ h' \longrightarrow h = h'$

lemma *strictly-exactD*:
$\llbracket$ *strictly-exact* P; $P\ h$; $P\ h'$ $\rrbracket \Longrightarrow h = h'$
⟨*proof*⟩

lemma *strictly-exactI*:
$(\bigwedge h\ h'.\ \llbracket P\ h;\ P\ h' \rrbracket \Longrightarrow h = h') \Longrightarrow$ *strictly-exact* P
⟨*proof*⟩

lemma *strictly-exact-sep-conj*:
$\llbracket$ *strictly-exact* P; *strictly-exact* Q $\rrbracket \Longrightarrow$ *strictly-exact* ($P \wedge* Q$)
⟨*proof*⟩

lemma *strictly-exact-conj-impl*:
$\llbracket$ ($Q \wedge*$ *sep-true*) h; $P\ h$; *strictly-exact* Q $\rrbracket \Longrightarrow (Q \wedge* (Q \longrightarrow* P))\ h$
⟨*proof*⟩

end

interpretation *sep*: *ab-semigroup-mult* (∗∗)
⟨*proof*⟩

interpretation *sep*: *comm-monoid-add* (∗∗) □
⟨*proof*⟩

5 Separation Algebra with Stronger, but More Intuitive Disjunction Axiom

class *stronger-sep-algebra* = *pre-sep-algebra* +
assumes *sep-add-disj-eq* [*simp*]: $y\ \#\#\ z \Longrightarrow x\ \#\#\ y + z = (x\ \#\#\ y \wedge x\ \#\#\ z)$
begin

lemma *sep-disj-add-eq* [*simp*]: $x\ \#\#\ y \Longrightarrow x + y\ \#\#\ z = (x\ \#\#\ z \wedge y\ \#\#\ z)$

⟨*proof*⟩

subclass *sep-algebra* ⟨*proof*⟩

end

6 Folding separating conjunction over lists of predicates

lemma *sep-list-conj-Nil* [*simp*]: $\bigwedge*$ [] = □
⟨*proof*⟩

lemma (**in** *semigroup-add*) *foldl-assoc*:
shows *foldl* (+) (*x*+*y*) *zs* = *x* + (*foldl* (+) *y zs*)
⟨*proof*⟩

lemma (**in** *monoid-add*) *foldl-absorb0*:
shows *x* + (*foldl* (+) *0 zs*) = *foldl* (+) *x zs*
⟨*proof*⟩

lemma *sep-list-conj-Cons* [*simp*]: $\bigwedge*$ (*x*#*xs*) = (*x* ∗∗ $\bigwedge*$ *xs*)
⟨*proof*⟩

lemma *sep-list-conj-append* [*simp*]: $\bigwedge*$ (*xs* @ *ys*) = ($\bigwedge*$ *xs* ∗∗ $\bigwedge*$ *ys*)
⟨*proof*⟩

lemma (**in** *comm-monoid-add*) *foldl-map-filter*:
foldl (+) *0* (*map f* (*filter P xs*)) +
foldl (+) *0* (*map f* (*filter* (*not P*) *xs*))
= *foldl* (+) *0* (*map f xs*)
⟨*proof*⟩

7 Separation Algebra with a Cancellative Monoid (for completeness)

Separation algebra with a cancellative monoid. The results of being a precise assertion (distributivity over separating conjunction) require this. although we never actually use this property in our developments, we keep it here for completeness.

class *cancellative-sep-algebra* = *sep-algebra* +
assumes *sep-add-cancelD*: ⟦ *x* + *z* = *y* + *z* ; *x* ## *z* ; *y* ## *z* ⟧ ⟹ *x* = *y*
begin

definition

$precise :: ('a \Rightarrow bool) \Rightarrow bool$ **where**
$precise\ P = (\forall h\ hp\ hp'.\ hp \preceq h \wedge P\ hp \wedge hp' \preceq h \wedge P\ hp' \longrightarrow hp = hp')$

lemma *precise* $((=)\ s)$
⟨*proof*⟩

lemma *sep-add-cancel*:
$x\ \#\#\ z \Longrightarrow y\ \#\#\ z \Longrightarrow (x + z = y + z) = (x = y)$
⟨*proof*⟩

lemma *precise-distribute*:
$precise\ P = (\forall Q\ R.\ ((Q\ and\ R) \wedge{*}\ P) = ((Q \wedge{*}\ P)\ and\ (R \wedge{*}\ P)))$
⟨*proof*⟩

lemma *strictly-precise*: $strictly\text{-}exact\ P \Longrightarrow precise\ P$
⟨*proof*⟩

end

end

8 Standard Heaps as an Instance of Separation Algebra

theory *Sep-Heap-Instance*
imports *Separation-Algebra*
begin

Example instantiation of a the separation algebra to a map, i.e. a function from any type to *'a option*.

class *opt* =
fixes *none* :: *'a*
begin
definition $domain\ f \equiv \{x.\ f\ x \neq none\}$
end

instantiation *option* :: (*type*) *opt*
begin
definition *none-def* [*simp*]: $none \equiv None$
instance ⟨*proof*⟩
end

instantiation *fun* :: (*type*, *opt*) *zero*
begin
definition *zero-fun-def*: $0 \equiv \lambda s.\ none$
instance ⟨*proof*⟩
end

instantiation *fun* :: (*type*, *opt*) *sep-algebra*
begin

definition
plus-fun-def: $m1 + m2 \equiv \lambda x.$ *if* $m2\ x = none$ *then* $m1\ x$ *else* $m2\ x$

definition
sep-disj-fun-def: *sep-disj* $m1\ m2 \equiv$ *domain* $m1 \cap$ *domain* $m2 = \{\}$

instance
⟨*proof*⟩

end

For the actual option type *domain* and + are just *dom* and ++:

lemma *domain-conv*: *domain* = *dom*
⟨*proof*⟩

lemma *plus-fun-conv*: $a + b = a ++ b$
⟨*proof*⟩

lemmas *map-convs* = *domain-conv plus-fun-conv*

Any map can now act as a separation heap without further work:

lemma
fixes *h* :: (*nat* => *nat*) => *'foo option*
shows (*P* ** *Q* ** *H*) *h* = (*Q* ** *H* ** *P*) *h*
⟨*proof*⟩

end

9 Separation Logic Tactics

theory *Sep-Tactics*
imports *Separation-Algebra*
begin

⟨*ML*⟩

A number of proof methods to assist with reasoning about separation logic.

10 Selection (move-to-front) tactics

⟨*ML*⟩

11 Substitution

⟨*ML*⟩

12 Forward Reasoning

⟨*ML*⟩

13 Backward Reasoning

⟨*ML*⟩

14 Cancellation of Common Conjuncts via Elimination Rules

named-theorems *sep-cancel*

The basic *sep-cancel-tac* is minimal. It only eliminates erule-derivable conjuncts between an assumption and the conclusion.

To have a more useful tactic, we augment it with more logic, to proceed as follows:

- try discharge the goal first using *tac*
- if that fails, invoke *sep-cancel-tac*
- if *sep-cancel-tac* succeeds
  - try to finish off with tac (but ok if that fails)
  - try to finish off with $\lambda s.$ *True* (but ok if that fails)

⟨*ML*⟩

As above, but use blast with a depth limit to figure out where cancellation can be done.

⟨*ML*⟩

end

15 Example from HOL/Hoare/Separation

theory *Simple-Separation-Example*
  imports *HOL−Hoare.Hoare-Logic-Abort ../Sep-Heap-Instance*
    ../Sep-Tactics
begin

declare [[*syntax-ambiguity-warning* = *false*]]

type-synonym *heap* = (*nat* ⇒ *nat option*)

definition *maps-to*:: *nat* ⇒ *nat* ⇒ *heap* ⇒ *bool* (‹- ↦ -› [*56*,*51*] *56*)
  where *x* ↦ *y* ≡ λ*h*. *h* = [*x* ↦ *y*]

notation *pred-ex* (**binder** ‹∃› *10*)

definition *maps-to-ex* :: *nat* ⇒ *heap* ⇒ *bool* (‹- ↦ −› [*56*] *56*)
  where *x* ↦ − ≡ ∃ *y*. *x* ↦ *y*

lemma *maps-to-maps-to-ex* [*elim*!]:
  (*p* ↦ *v*) *s* ⟹ (*p* ↦ −) *s*
  ⟨*proof*⟩

lemma *maps-to-write*:
  (*p* ↦ − ∗∗ *P*) *H* ⟹ (*p* ↦ *v* ∗∗ *P*) (*H* (*p* ↦ *v*))
  ⟨*proof*⟩

lemma *points-to*:
  (*p* ↦ *v* ∗∗ *P*) *H* ⟹ *the* (*H p*) = *v*
  ⟨*proof*⟩

primrec
  list :: *nat* ⇒ *nat list* ⇒ *heap* ⇒ *bool*
where
  list i [] = (⟨*i*=*0*⟩ *and* ☐)
| *list i* (*x*#*xs*) = (⟨*i*=*x* ∧ *i*≠*0*⟩ *and* (*EXS j*. *i* ↦ *j* ∗∗ *list j xs*))

lemma *list-empty* [*simp*]:
  shows *list 0 xs* = (λ*s*. *xs* = [] ∧ ☐ *s*)
  ⟨*proof*⟩

lemma *VARS x y z w h*
 {(*x* ↦ *y* ∗∗ *z* ↦ *w*) *h*}
 SKIP
 {*x* ≠ *z*}
  ⟨*proof*⟩

lemma *VARS H x y z w*

{$(P ** Q)\ H$}
SKIP
{$(Q ** P)\ H$}
⟨*proof*⟩

lemma *VARS H*
{$p \neq 0 \land (p \mapsto - ** list\ q\ qs)\ H$}
$H := H(p \mapsto q)$
{$list\ p\ (p\#qs)\ H$}
⟨*proof*⟩

lemma *VARS H p q r*
{$(list\ p\ Ps ** list\ q\ Qs)\ H$}
WHILE $p \neq 0$
INV {$\exists ps\ qs.\ (list\ p\ ps ** list\ q\ qs)\ H \land rev\ ps\ @\ qs = rev\ Ps\ @\ Qs$}
DO $r := p$; $p := the(H\ p)$; $H := H(r \mapsto q)$; $q := r$ *OD*
{$list\ q\ (rev\ Ps\ @\ Qs)\ H$}
⟨*proof*⟩

end

theory *Sep-Tactics-Test*
imports ../*Sep-Tactics*
begin

Substitution and forward/backward reasoning

typedecl *p*
typedecl *val*
typedecl *heap*

axiomatization where *heap-sep-algebra*: *OFCLASS*(*heap*, *sep-algebra-class*)
instance *heap* :: *sep-algebra* ⟨*proof*⟩

axiomatization
points-to :: $p \Rightarrow val \Rightarrow heap \Rightarrow bool$ **and**
val :: $heap \Rightarrow p \Rightarrow val$
where
points-to: $(points\text{-}to\ p\ v ** P)\ h \Longrightarrow val\ h\ p = v$

lemma
$[\![\ Q2\ (val\ h\ p);\ (K ** T ** blub ** P ** points\text{-}to\ p\ v ** P ** J)\ h\ ]\!]$
$\Longrightarrow Q\ (val\ h\ p)\ (val\ h\ p)$
⟨*proof*⟩

lemma
$[\![\ Q2\ (val\ h\ p);\ (K ** T ** blub ** P ** points\text{-}to\ p\ v ** P ** J)\ h\ ]\!]$
$\Longrightarrow Q\ (val\ h\ p)\ (val\ h\ p)$

⟨*proof*⟩

lemma
⟦ *Q2* (*val h p*); (*K* ∗∗ *T* ∗∗ *blub* ∗∗ *P* ∗∗ *points-to p v* ∗∗ *P* ∗∗ *J*) *h* ⟧
⟹ *Q* (*val h p*) (*val h p*)
⟨*proof*⟩

consts
update :: *p* ⇒ *val* ⇒ *heap* ⇒ *heap*

schematic-goal
assumes *a*: ⋀*P*. (*stuff p* ∗∗ *P*) *H* ⟹ (*other-stuff p v* ∗∗ *P*) (*update p v H*)
shows (*X* ∗∗ *Y* ∗∗ *other-stuff p ?v*) (*update p v H*)
⟨*proof*⟩

Example of low-level rewrites

lemma ⟦ *unrelated s* ; (*P* ∗∗ *Q* ∗∗ *R*) *s* ⟧ ⟹ (*A* ∗∗ *B* ∗∗ *Q* ∗∗ *P*) *s*
⟨*proof*⟩

Conjunct selection

lemma (*A* ∗∗ *B* ∗∗ *Q* ∗∗ *P*) *s*
⟨*proof*⟩

lemma ⟦ *also unrelated*; (*A* ∗∗ *B* ∗∗ *Q* ∗∗ *P*) *s* ⟧ ⟹ *unrelated*
⟨*proof*⟩

16 Test cases for *sep-cancel*.

lemma
assumes *forward*: ⋀*s g p v*. *A g p v s* ⟹ *AA g p s*
shows ⋀*xv yv P s y x s*. (*A g x yv* ∗∗ *A g y yv* ∗∗ *P*) *s* ⟹ (*AA g y* ∗∗ *sep-true*) *s*
⟨*proof*⟩

lemma
assumes *forward*: ⋀*s*. *generic s* ⟹ *instance s*
shows (*A* ∗∗ *generic* ∗∗ *B*) *s* ⟹ (*instance* ∗∗ *sep-true*) *s*
⟨*proof*⟩

lemma ⟦ (*A* ∗∗ *B*) *sa* ; (*A* ∗∗ *Y*) *s* ⟧ ⟹ (*A* ∗∗ *X*) *s*
⟨*proof*⟩

lemma ⟦ (*A* ∗∗ *B*) *sa* ; (*A* ∗∗ *Y*) *s* ⟧ ⟹ (λ*s*. (*A* ∗∗ *X*) *s*) *s*
⟨*proof*⟩

schematic-goal ⟦ (*B* ∗∗ *A* ∗∗ *C*) *s* ⟧ ⟹ (λ*s*. (*A* ∗∗ *?X*) *s*) *s*
⟨*proof*⟩

lemma
  assumes *forward*: $\bigwedge s.$ *generic s* $\Longrightarrow$ *instance s*
  shows $[\![$ (*A* $**$ *B*) *s* ; (*generic* $**$ *Y*) *s* $]\!]$ $\Longrightarrow$ (*X* $**$ *instance*) *s*
  ⟨*proof*⟩

lemma
  assumes *forward*: $\bigwedge s.$ *generic s* $\Longrightarrow$ *instance s*
  shows *generic s* $\Longrightarrow$ *instance s*
  ⟨*proof*⟩

lemma
  assumes *forward*: $\bigwedge s.$ *generic s* $\Longrightarrow$ *instance s*
  assumes *forward2*: $\bigwedge s.$ *instance s* $\Longrightarrow$ *instance2 s*
  shows *generic s* $\Longrightarrow$ (*instance2* $**$ *sep-true*) *s*
  ⟨*proof*⟩

end

17 More properties of maps plus map disjuction.

theory *Map-Extra*
  imports *Main*
begin

A note on naming: Anything not involving heap disjuction can potentially be incorporated directly into Map.thy, thus uses m for map variable names. Anything involving heap disjunction is not really mergeable with Map, is destined for use in separation logic, and hence uses h

18 Things that could go into Option Type

Misc option lemmas

lemma *None-not-eq*: (*None* $\neq$ *x*) = ($\exists$ *y*. *x* = *Some y*) ⟨*proof*⟩

lemma *None-com*: (*None* = *x*) = (*x* = *None*) ⟨*proof*⟩

lemma *Some-com*: (*Some y* = *x*) = (*x* = *Some y*) ⟨*proof*⟩

19 Things that go into Map.thy

Map intersection: set of all keys for which the maps agree.

definition
  map-inter :: ($'a \rightharpoonup 'b$) $\Rightarrow$ ($'a \rightharpoonup 'b$) $\Rightarrow$ $'a$ *set* (**infixl** ‹$\cap_m$› *70*) **where**
  $m_1 \cap_m m_2 \equiv \{x \in dom\ m_1.\ m_1\ x = m_2\ x\}$

Map restriction via domain subtraction

definition
sub-restrict-map :: $('a \rightharpoonup 'b)$ => *'a set* => $('a \rightharpoonup 'b)$ (**infixl** ‹‘—› *110*)
where
m ‘— $S \equiv (\lambda x.\ if\ x \in S\ then\ None\ else\ m\ x)$

19.1 Properties of maps not related to restriction

lemma *empty-forall-equiv*: $(m = Map.empty) = (\forall x.\ m\ x = None)$
⟨*proof*⟩

lemma *map-le-empty2* [*simp*]:
$(m \subseteq_m Map.empty) = (m = Map.empty)$
⟨*proof*⟩

lemma *dom-iff*:
$(\exists y.\ m\ x = Some\ y) = (x \in dom\ m)$
⟨*proof*⟩

lemma *non-dom-eval*:
$x \notin dom\ m \Longrightarrow m\ x = None$
⟨*proof*⟩

lemma *non-dom-eval-eq*:
$x \notin dom\ m = (m\ x = None)$
⟨*proof*⟩

lemma *map-add-same-left-eq*:
$m_1 = m_1{}' \Longrightarrow (m_0 ++ m_1 = m_0 ++ m_1{}')$
⟨*proof*⟩

lemma *map-add-left-cancelI* [*intro*!]:
$m_1 = m_1{}' \Longrightarrow m_0 ++ m_1 = m_0 ++ m_1{}'$
⟨*proof*⟩

lemma *dom-empty-is-empty*:
$(dom\ m = \{\}) = (m = Map.empty)$
⟨*proof*⟩

lemma *map-add-dom-eq*:
$dom\ m = dom\ m' \Longrightarrow m ++ m' = m'$
⟨*proof*⟩

lemma *map-add-right-dom-eq*:
$\llbracket\ m_0 ++ m_1 = m_0{}' ++ m_1{}';\ dom\ m_1 = dom\ m_1{}'\ \rrbracket \Longrightarrow m_1 = m_1{}'$
⟨*proof*⟩

lemma *map-le-same-dom-eq*:
$\llbracket\ m_0 \subseteq_m m_1\ ;\ dom\ m_0 = dom\ m_1\ \rrbracket \Longrightarrow m_0 = m_1$
⟨*proof*⟩

19.2 Properties of map restriction

lemma *restrict-map-cancel*:
$(m \restriction\text{`}\ S = m \restriction\text{`}\ T) = (dom\ m \cap S = dom\ m \cap T)$
⟨*proof*⟩

lemma *map-add-restricted-self* [*simp*]:
$m ++ m \restriction\text{`}\ S = m$
⟨*proof*⟩

lemma *map-add-restrict-dom-right* [*simp*]:
$(m ++ m') \restriction\text{`}\ dom\ m' = m'$
⟨*proof*⟩

lemma *restrict-map-UNIV* [*simp*]:
$m \restriction\text{`}\ UNIV = m$
⟨*proof*⟩

lemma *restrict-map-dom*:
$S = dom\ m \Longrightarrow m \restriction\text{`}\ S = m$
⟨*proof*⟩

lemma *restrict-map-subdom*:
$dom\ m \subseteq S \Longrightarrow m \restriction\text{`}\ S = m$
⟨*proof*⟩

lemma *map-add-restrict*:
$(m_0 ++ m_1) \restriction\text{`}\ S = ((m_0 \restriction\text{`}\ S) ++ (m_1 \restriction\text{`}\ S))$
⟨*proof*⟩

lemma *map-le-restrict*:
$m \subseteq_m m' \Longrightarrow m = m' \restriction\text{`}\ dom\ m$
⟨*proof*⟩

lemma *restrict-map-le*:
$m \restriction\text{`}\ S \subseteq_m m$
⟨*proof*⟩

lemma *restrict-map-remerge*:
$[\![\ S \cap T = \{\}\ ]\!] \Longrightarrow m \restriction\text{`}\ S ++ m \restriction\text{`}\ T = m \restriction\text{`}\ (S \cup T)$
⟨*proof*⟩

lemma *restrict-map-empty*:
$dom\ m \cap S = \{\} \Longrightarrow m \restriction\text{`}\ S = Map.empty$
⟨*proof*⟩

lemma *map-add-restrict-comp-right* [*simp*]:
$(m \restriction\text{`}\ S ++ m \restriction\text{`}\ (UNIV - S)) = m$
⟨*proof*⟩

lemma *map-add-restrict-comp-right-dom* [*simp*]:
$(m \restriction_{\text{`}} S ++ m \restriction_{\text{`}} (dom\ m - S)) = m$
⟨*proof*⟩

lemma *map-add-restrict-comp-left* [*simp*]:
$(m \restriction_{\text{`}} (UNIV - S) ++ m \restriction_{\text{`}} S) = m$
⟨*proof*⟩

lemma *restrict-self-UNIV*:
$m \restriction_{\text{`}} (dom\ m - S) = m \restriction_{\text{`}} (UNIV - S)$
⟨*proof*⟩

lemma *map-add-restrict-nonmember-right*:
$x \notin dom\ m' \Longrightarrow (m ++ m') \restriction_{\text{`}} \{x\} = m \restriction_{\text{`}} \{x\}$
⟨*proof*⟩

lemma *map-add-restrict-nonmember-left*:
$x \notin dom\ m \Longrightarrow (m ++ m') \restriction_{\text{`}} \{x\} = m' \restriction_{\text{`}} \{x\}$
⟨*proof*⟩

lemma *map-add-restrict-right*:
$x \subseteq dom\ m' \Longrightarrow (m ++ m') \restriction_{\text{`}} x = m' \restriction_{\text{`}} x$
⟨*proof*⟩

lemma *restrict-map-compose*:
$[\![S \cup T = dom\ m ; S \cap T = \{\}]\!] \Longrightarrow m \restriction_{\text{`}} S ++ m \restriction_{\text{`}} T = m$
⟨*proof*⟩

lemma *map-le-dom-subset-restrict*:
$[\![m' \subseteq_m m; dom\ m' \subseteq S]\!] \Longrightarrow m' \subseteq_m (m \restriction_{\text{`}} S)$
⟨*proof*⟩

lemma *map-le-dom-restrict-sub-add*:
$m' \subseteq_m m \Longrightarrow m \restriction_{\text{`}} (dom\ m - dom\ m') ++ m' = m$
⟨*proof*⟩

lemma *subset-map-restrict-sub-add*:
$T \subseteq S \Longrightarrow m \restriction_{\text{`}} (S - T) ++ m \restriction_{\text{`}} T = m \restriction_{\text{`}} S$
⟨*proof*⟩

lemma *restrict-map-sub-union*:
$m \restriction_{\text{`}} (dom\ m - (S \cup T)) = (m \restriction_{\text{`}} (dom\ m - T)) \restriction_{\text{`}} (dom\ m - S)$
⟨*proof*⟩

lemma *prod-restrict-map-add*:
$[\![S \cup T = U; S \cap T = \{\}]\!] \Longrightarrow m \restriction_{\text{`}} (X \times S) ++ m \restriction_{\text{`}} (X \times T) = m \restriction_{\text{`}} (X \times U)$
⟨*proof*⟩

20 Things that should not go into Map.thy (separation logic)

20.1 Definitions

Map disjuction

definition
map-disj :: $('a \rightharpoonup 'b) \Rightarrow ('a \rightharpoonup 'b) \Rightarrow bool$ (**infix** ‹⊥› *51*) **where**
$h_0 \perp h_1 \equiv dom\ h_0 \cap dom\ h_1 = \{\}$

declare *None-not-eq* [*simp*]

20.2 Properties of (‘−)

lemma *restrict-map-sub-disj*: $h\ |`\ S \perp h\ `-\ S$
⟨*proof*⟩

lemma *restrict-map-sub-add*: $h\ |`\ S\ ++\ h\ `-\ S = h$
⟨*proof*⟩

20.3 Properties of map disjunction

lemma *map-disj-empty-right* [*simp*]:
$h \perp Map.empty$
⟨*proof*⟩

lemma *map-disj-empty-left* [*simp*]:
$Map.empty \perp h$
⟨*proof*⟩

lemma *map-disj-com*:
$h_0 \perp h_1 = h_1 \perp h_0$
⟨*proof*⟩

lemma *map-disjD*:
$h_0 \perp h_1 \Longrightarrow dom\ h_0 \cap dom\ h_1 = \{\}$
⟨*proof*⟩

lemma *map-disjI*:
$dom\ h_0 \cap dom\ h_1 = \{\} \Longrightarrow h_0 \perp h_1$
⟨*proof*⟩

20.4 Map associativity-commutativity based on map disjuction

lemma *map-add-com*:
$h_0 \perp h_1 \Longrightarrow h_0\ ++\ h_1 = h_1\ ++\ h_0$
⟨*proof*⟩

lemma *map-add-left-commute*:
$h_0 \perp h_1 \Longrightarrow h_0 ++ (h_1 ++ h_2) = h_1 ++ (h_0 ++ h_2)$
⟨*proof*⟩

lemma *map-add-disj*:
$h_0 \perp (h_1 ++ h_2) = (h_0 \perp h_1 \wedge h_0 \perp h_2)$
⟨*proof*⟩

lemma *map-add-disj'*:
$(h_1 ++ h_2) \perp h_0 = (h_1 \perp h_0 \wedge h_2 \perp h_0)$
⟨*proof*⟩

We redefine (++) associativity to bind to the right, which seems to be the more common case. Note that when a theory includes Map again, *map-add-assoc* will return to the simpset and will cause infinite loops if its symmetric counterpart is added (e.g. via *map-add-ac*)

declare *map-add-assoc* [*simp del*]

Since the associativity-commutativity of (++) relies on map disjunction, we include some basic rules into the ac set.

lemmas *map-add-ac* =
map-add-assoc[*symmetric*] *map-add-com map-disj-com*
map-add-left-commute map-add-disj map-add-disj'

20.5 Basic properties

lemma *map-disj-None-right*:
$[\![h_0 \perp h_1 ; x \in dom\ h_0]\!] \Longrightarrow h_1\ x = None$
⟨*proof*⟩

lemma *map-disj-None-left*:
$[\![h_0 \perp h_1 ; x \in dom\ h_1]\!] \Longrightarrow h_0\ x = None$
⟨*proof*⟩

lemma *map-disj-None-left'*:
$[\![h_0\ x = Some\ y ; h_1 \perp h_0]\!] \Longrightarrow h_1\ x = None$
⟨*proof*⟩

lemma *map-disj-None-right'*:
$[\![h_1\ x = Some\ y ; h_1 \perp h_0]\!] \Longrightarrow h_0\ x = None$
⟨*proof*⟩

lemma *map-disj-common*:
$[\![h_0 \perp h_1 ; h_0\ p = Some\ v ; h_1\ p = Some\ v']\!] \Longrightarrow False$
⟨*proof*⟩

lemma *map-disj-eq-dom-left*:
$[\![h_0 \perp h_1 ; dom\ h_0{}' = dom\ h_0]\!] \Longrightarrow h_0{}' \perp h_1$
⟨*proof*⟩

20.6 Map disjunction and addition

lemma *map-add-eval-left*:
$[\![\ x \in dom\ h\ ;\ h \perp h'\ ]\!] \Longrightarrow (h ++ h')\ x = h\ x$
⟨*proof*⟩

lemma *map-add-eval-right*:
$[\![\ x \in dom\ h'\ ;\ h \perp h'\ ]\!] \Longrightarrow (h ++ h')\ x = h'\ x$
⟨*proof*⟩

lemma *map-add-eval-left'*:
$[\![\ x \notin dom\ h'\ ;\ h \perp h'\ ]\!] \Longrightarrow (h ++ h')\ x = h\ x$
⟨*proof*⟩

lemma *map-add-eval-right'*:
$[\![\ x \notin dom\ h\ ;\ h \perp h'\ ]\!] \Longrightarrow (h ++ h')\ x = h'\ x$
⟨*proof*⟩

lemma *map-add-left-dom-eq*:
assumes *eq*: $h_0 ++ h_1 = h_0' ++ h_1'$
assumes *etc*: $h_0 \perp h_1\ h_0' \perp h_1'\ dom\ h_0 = dom\ h_0'$
shows $h_0 = h_0'$
⟨*proof*⟩

lemma *map-add-left-eq*:
assumes *eq*: $h_0 ++ h = h_1 ++ h$
assumes *disj*: $h_0 \perp h\ h_1 \perp h$
shows $h_0 = h_1$
⟨*proof*⟩

lemma *map-add-right-eq*:
$[\![h ++ h_0 = h ++ h_1;\ h_0 \perp h;\ h_1 \perp h]\!] \Longrightarrow h_0 = h_1$
⟨*proof*⟩

lemma *map-disj-add-eq-dom-right-eq*:
assumes *merge*: $h_0 ++ h_1 = h_0' ++ h_1'$ **and** *d*: $dom\ h_0 = dom\ h_0'$ **and**
ab-disj: $h_0 \perp h_1$ **and** *cd-disj*: $h_0' \perp h_1'$
shows $h_1 = h_1'$
⟨*proof*⟩

lemma *map-disj-add-eq-dom-left-eq*:
assumes *add*: $h_0 ++ h_1 = h_0' ++ h_1'$ **and**
dom: $dom\ h_1 = dom\ h_1'$ **and**
disj: $h_0 \perp h_1\ h_0' \perp h_1'$
shows $h_0 = h_0'$
⟨*proof*⟩

lemma *map-add-left-cancel*:
assumes *disj*: $h_0 \perp h_1\ h_0 \perp h_1'$
shows $(h_0 ++ h_1 = h_0 ++ h_1') = (h_1 = h_1')$

$\langle proof \rangle$

lemma *map-add-lr-disj*:
$[\![h_0 ++ h_1 = h_0' ++ h_1'; h_1 \perp h_1']\!] \Longrightarrow dom\ h_1 \subseteq dom\ h_0'$
$\langle proof \rangle$

20.7 Map disjunction and map updates

lemma *map-disj-update-left* [*simp*]:
$p \in dom\ h_1 \Longrightarrow h_0 \perp h_1(p \mapsto v) = h_0 \perp h_1$
$\langle proof \rangle$

lemma *map-disj-update-right* [*simp*]:
$p \in dom\ h_1 \Longrightarrow h_1(p \mapsto v) \perp h_0 = h_1 \perp h_0$
$\langle proof \rangle$

lemma *map-add-update-left*:
$[\![h_0 \perp h_1 ; p \in dom\ h_0]\!] \Longrightarrow (h_0 ++ h_1)(p \mapsto v) = (h_0(p \mapsto v) ++ h_1)$
$\langle proof \rangle$

lemma *map-add-update-right*:
$[\![h_0 \perp h_1 ; p \in dom\ h_1]\!] \Longrightarrow (h_0 ++ h_1)(p \mapsto v) = (h_0 ++ h_1\ (p \mapsto v))$
$\langle proof \rangle$

lemma *map-add3-update*:
$[\![h_0 \perp h_1 ; h_1 \perp h_2 ; h_0 \perp h_2 ; p \in dom\ h_0]\!]$
$\Longrightarrow (h_0 ++ h_1 ++ h_2)(p \mapsto v) = h_0(p \mapsto v) ++ h_1 ++ h_2$
$\langle proof \rangle$

20.8 Map disjunction and $(\subseteq_m)$

lemma *map-le-override* [*simp*]:
$[\![h \perp h']\!] \Longrightarrow h \subseteq_m h ++ h'$
$\langle proof \rangle$

lemma *map-leI-left*:
$[\![h = h_0 ++ h_1 ; h_0 \perp h_1]\!] \Longrightarrow h_0 \subseteq_m h$ $\langle proof \rangle$

lemma *map-leI-right*:
$[\![h = h_0 ++ h_1 ; h_0 \perp h_1]\!] \Longrightarrow h_1 \subseteq_m h$ $\langle proof \rangle$

lemma *map-disj-map-le*:
$[\![h_0' \subseteq_m h_0; h_0 \perp h_1]\!] \Longrightarrow h_0' \perp h_1$
$\langle proof \rangle$

lemma *map-le-on-disj-left*:
$[\![h' \subseteq_m h ; h_0 \perp h_1 ; h' = h_0 ++ h_1]\!] \Longrightarrow h_0 \subseteq_m h$
$\langle proof \rangle$

lemma *map-le-on-disj-right*:

$[\![h' \subseteq_m h ; h_0 \perp h_1 ; h' = h_1 ++ h_0]\!] \Longrightarrow h_0 \subseteq_m h$
⟨*proof*⟩

lemma *map-le-add-cancel*:
$[\![h_0 \perp h_1 ; h_0' \subseteq_m h_0]\!] \Longrightarrow h_0' ++ h_1 \subseteq_m h_0 ++ h_1$
⟨*proof*⟩

lemma *map-le-override-bothD*:
assumes *subm*: $h_0' ++ h_1 \subseteq_m h_0 ++ h_1$
assumes *disj'*: $h_0' \perp h_1$
assumes *disj*: $h_0 \perp h_1$
shows $h_0' \subseteq_m h_0$
⟨*proof*⟩

lemma *map-le-conv*:
$(h_0' \subseteq_m h_0 \wedge h_0' \neq h_0) = (\exists h_1.\ h_0 = h_0' ++ h_1 \wedge h_0' \perp h_1 \wedge h_0' \neq h_0)$
⟨*proof*⟩

lemma *map-le-conv2*:
$h_0' \subseteq_m h_0 = (\exists h_1.\ h_0 = h_0' ++ h_1 \wedge h_0' \perp h_1)$
⟨*proof*⟩

20.9 Map disjunction and restriction

lemma *map-disj-comp* [*simp*]:
$h_0 \perp h_1 |` (UNIV - dom\ h_0)$
⟨*proof*⟩

lemma *restrict-map-disj*:
$S \cap T = \{\} \Longrightarrow h |` S \perp h |` T$
⟨*proof*⟩

lemma *map-disj-restrict-dom* [*simp*]:
$h_0 \perp h_1 |` (dom\ h_1 - dom\ h_0)$
⟨*proof*⟩

lemma *restrict-map-disj-dom-empty*:
$h \perp h' \Longrightarrow h |` dom\ h' = Map.empty$
⟨*proof*⟩

lemma *restrict-map-univ-disj-eq*:
$h \perp h' \Longrightarrow h |` (UNIV - dom\ h') = h$
⟨*proof*⟩

lemma *restrict-map-disj-dom*:
$h_0 \perp h_1 \Longrightarrow h |` dom\ h_0 \perp h |` dom\ h_1$
⟨*proof*⟩

lemma *map-add-restrict-dom-left*:

$h \perp h' \Longrightarrow (h ++ h') |` \, dom \; h = h$
⟨proof⟩

lemma *map-add-restrict-dom-left'*:
$h \perp h' \Longrightarrow S = dom \; h \Longrightarrow (h ++ h') |` \, S = h$
⟨proof⟩

lemma *restrict-map-disj-left*:
$h_0 \perp h_1 \Longrightarrow h_0 |` \, S \perp h_1$
⟨proof⟩

lemma *restrict-map-disj-right*:
$h_0 \perp h_1 \Longrightarrow h_0 \perp h_1 |` \, S$
⟨proof⟩

lemmas *restrict-map-disj-both* = *restrict-map-disj-right restrict-map-disj-left*

lemma *map-dom-disj-restrict-right*:
$h_0 \perp h_1 \Longrightarrow (h_0 ++ h_0') |` \, dom \; h_1 = h_0' |` \, dom \; h_1$
⟨proof⟩

lemma *restrict-map-on-disj*:
$h_0' \perp h_1 \Longrightarrow h_0 |` \, dom \; h_0' \perp h_1$
⟨proof⟩

lemma *restrict-map-on-disj'*:
$h_0 \perp h_1 \Longrightarrow h_0 \perp h_1 |` \, S$
⟨proof⟩

lemma *map-le-sub-dom*:
$[\![h_0 ++ h_1 \subseteq_m h \, ; h_0 \perp h_1]\!] \Longrightarrow h_0 \subseteq_m h |` \, (dom \; h - dom \; h_1)$
⟨proof⟩

lemma *map-submap-break*:
$[\![h \subseteq_m h']\!] \Longrightarrow h' = (h' |` \, (UNIV - dom \; h)) ++ h$
⟨proof⟩

lemma *map-add-disj-restrict-both*:
$[\![h_0 \perp h_1; S \cap S' = \{\}; T \cap T' = \{\}]\!]$
$\Longrightarrow (h_0 |` \, S) ++ (h_1 |` \, T) \perp (h_0 |` \, S') ++ (h_1 |` \, T')$
⟨proof⟩

end

21 Separation Algebra for Virtual Memory

theory *VM-Example*
imports *../Sep-Tactics ../Map-Extra*
begin

Example instantiation of the abstract separation algebra to the sliced-memory model used for building a separation logic in "Verification of Programs in Virtual Memory Using Separation Logic" (PhD Thesis) by Rafal Kolanski.

We wrap up the concept of physical and virtual pointers as well as value (usually a byte), and the page table root, into a datatype for instantiation. This avoids having to produce a hierarchy of type classes.

The result is more general than the original. It does not mention the types of pointers or virtual memory addresses. Instead of supporting only singleton page table roots, we now support sets so we can identify a single 0 for the monoid. This models multiple page tables in memory, whereas the original logic was only capable of one at a time.

```
datatype ('p,'v,'value,'r) vm-sep-state
  = VMSepState ((('p × 'v) ⇀ 'value) × 'r set)

instantiation vm-sep-state :: (type, type, type, type) sep-algebra
begin

fun
  vm-heap :: ('a,'b,'c,'d) vm-sep-state ⇒ (('a × 'b) ⇀ 'c) where
  vm-heap (VMSepState (h,r)) = h

fun
  vm-root :: ('a,'b,'c,'d) vm-sep-state ⇒ 'd set where
  vm-root (VMSepState (h,r)) = r

definition
  sep-disj-vm-sep-state :: ('a, 'b, 'c, 'd) vm-sep-state
                        ⇒ ('a, 'b, 'c, 'd) vm-sep-state ⇒ bool where
  sep-disj-vm-sep-state x y = vm-heap x ⊥ vm-heap y

definition
  zero-vm-sep-state :: ('a, 'b, 'c, 'd) vm-sep-state where
  zero-vm-sep-state ≡ VMSepState (Map.empty, {})

fun
  plus-vm-sep-state :: ('a, 'b, 'c, 'd) vm-sep-state
                    ⇒ ('a, 'b, 'c, 'd) vm-sep-state
                    ⇒ ('a, 'b, 'c, 'd) vm-sep-state where
  plus-vm-sep-state (VMSepState (x,r)) (VMSepState (y,r'))
    = VMSepState (x ++ y, r ∪ r')

instance
  ⟨proof⟩

end
```

end

22 Abstract Separation Logic, Alternative Definition

theory *Separation-Algebra-Alt*
imports *Main*
begin

This theory contains an alternative definition of speration algebra, following Calcagno et al very closely. While some of the abstract development is more algebraic, it is cumbersome to instantiate. We only use it to prove equivalence and to give an impression of how it would look like.

no-notation *map-add* (**infixl** ‹++› *100*)

definition
  lift2 :: (*'a => 'b => 'c option*) ⇒ *'a option => 'b option => 'c option*
where
  lift2 f a b ≡ *case* (*a*,*b*) *of* (*Some a*, *Some b*) ⇒ *f a b* | - ⇒ *None*

class *sep-algebra-alt* = *zero* +
  fixes *add* :: *'a => 'a => 'a option* (**infixr** ‹⊕› *65*)

  assumes *add-zero* [*simp*]: *x* ⊕ *0* = *Some x*
  assumes *add-comm*: *x* ⊕ *y* = *y* ⊕ *x*
  assumes *add-assoc*: *lift2 add a* (*lift2 add b c*) = *lift2 add* (*lift2 add a b*) *c*

begin

definition
  disjoint :: *'a => 'a => bool* (**infix** ‹##› *60*)
where
  a ## *b* ≡ *a* ⊕ *b* ≠ *None*

lemma *disj-com*: *x* ## *y* = *y* ## *x*
  ⟨*proof*⟩

lemma *disj-zero* [*simp*]: *x* ## *0*
  ⟨*proof*⟩

lemma *disj-zero2* [*simp*]: *0* ## *x*
  ⟨*proof*⟩

lemma *add-zero2* [*simp*]: *0* ⊕ *x* = *Some x*
  ⟨*proof*⟩

definition

substate :: *'a* => *'a* => *bool* (**infix** ‹$\preceq$› *60*) **where**
$a \preceq b \equiv \exists c.\ a \oplus c = Some\ b$

definition
sep-conj :: (*'a* ⇒ *bool*) ⇒ (*'a* ⇒ *bool*) ⇒ (*'a* ⇒ *bool*) (**infixl** ‹∗∗› *61*)
where
$P \ast\ast\ Q \equiv \lambda s.\ \exists p\ q.\ p \oplus q = Some\ s \wedge P\ p \wedge Q\ q$

definition *emp* :: *'a* ⇒ *bool* (‹□›) **where**
$\square \equiv \lambda s.\ s = 0$

definition
sep-impl :: (*'a* ⇒ *bool*) ⇒ (*'a* ⇒ *bool*) ⇒ (*'a* ⇒ *bool*) (**infixr** ‹⟶*› *25*)
where
$P \longrightarrow^* Q \equiv \lambda h.\ \forall h'\ h''.\ h \oplus h' = Some\ h'' \wedge P\ h' \longrightarrow Q\ h''$

definition (**in** −)
sep-true ≡ λ*s*. *True*

definition (**in** −)
sep-false ≡ λ*s*. *False*

abbreviation
add2 :: *'a option* => *'a option* => *'a option* (**infixr** ‹++› *65*)
where
add2 == *lift2 add*

lemma *add2-comm*:
a ++ *b* = *b* ++ *a*
⟨*proof*⟩

lemma *add2-None* [*simp*]:
x ++ *None* = *None*
⟨*proof*⟩

lemma *None-add2* [*simp*]:
None ++ *x* = *None*
⟨*proof*⟩

lemma *add2-Some-Some*:
Some x ++ *Some y* = *x* ⊕ *y*
⟨*proof*⟩

lemma *add2-zero* [*simp*]:
Some x ++ *Some 0* = *Some x*
⟨*proof*⟩

lemma *zero-add2* [*simp*]:
  $Some\ 0\ ++\ Some\ x = Some\ x$
  ⟨*proof*⟩

lemma *sep-conjE*:
  $[\![\ (P\ **\ Q)\ s;\ \bigwedge p\ q.\ [\![\ P\ p;\ Q\ q;\ p \oplus q = Some\ s\ ]\!] \Longrightarrow X\ ]\!] \Longrightarrow X$
  ⟨*proof*⟩

lemma *sep-conjI*:
  $[\![\ P\ p;\ Q\ q;\ p \oplus q = Some\ s\ ]\!] \Longrightarrow (P\ **\ Q)\ s$
  ⟨*proof*⟩

lemma *sep-conj-comI*:
  $(P\ **\ Q)\ s \Longrightarrow (Q\ **\ P)\ s$
  ⟨*proof*⟩

lemma *sep-conj-com*:
  $P\ **\ Q = Q\ **\ P$
  ⟨*proof*⟩

lemma *lift-to-add2*:
  $[\![z \oplus q = Some\ s;\ x \oplus y = Some\ q]\!] \Longrightarrow Some\ z\ ++\ Some\ x\ ++\ Some\ y = Some\ s$
  ⟨*proof*⟩

lemma *lift-to-add2'*:
  $[\![q \oplus z = Some\ s;\ x \oplus y = Some\ q]\!] \Longrightarrow (Some\ x\ ++\ Some\ y)\ ++\ Some\ z = Some\ s$
  ⟨*proof*⟩

lemma *add2-Some*:
  $(x\ ++\ Some\ y = Some\ z) = (\exists x'.\ x = Some\ x' \wedge x' \oplus y = Some\ z)$
  ⟨*proof*⟩

lemma *Some-add2*:
  $(Some\ x\ ++\ y = Some\ z) = (\exists y'.\ y = Some\ y' \wedge x \oplus y' = Some\ z)$
  ⟨*proof*⟩

lemma *sep-conj-assoc*:
  $P\ **\ (Q\ **\ R) = (P\ **\ Q)\ **\ R$
  ⟨*proof*⟩

lemma (**in** −) *sep-true*[*simp*]: *sep-true s* ⟨*proof*⟩
lemma (**in** −) *sep-false*[*simp*]: ¬*sep-false x* ⟨*proof*⟩

lemma *sep-conj-sep-true*:
  $P\ s \Longrightarrow (P\ **\ sep\text{-}true)\ s$
  ⟨*proof*⟩

lemma *sep-conj-sep-true'*:
$P\ s \Longrightarrow (sep\text{-}true ** P)\ s$
⟨*proof*⟩

lemma *disjoint-submaps-exist*:
$\exists\, h_0\ h_1.\ h_0 \oplus h_1 = Some\ h$
⟨*proof*⟩

lemma *sep-conj-true*[*simp*]:
$(sep\text{-}true ** sep\text{-}true) = sep\text{-}true$
⟨*proof*⟩

lemma *sep-conj-false-right*[*simp*]:
$(P ** sep\text{-}false) = sep\text{-}false$
⟨*proof*⟩

lemma *sep-conj-false-left*[*simp*]:
$(sep\text{-}false ** P) = sep\text{-}false$
⟨*proof*⟩

lemma *sep-conj-left-com*:
$(P ** (Q ** R)) = (Q ** (P ** R))$ (**is** $?x = ?y$)
⟨*proof*⟩

lemmas *sep-conj-ac* = *sep-conj-com sep-conj-assoc sep-conj-left-com*

lemma *empty-empty*[*simp*]: $\square\ 0$
⟨*proof*⟩

lemma *sep-conj-empty*[*simp*]:
$(P ** \square) = P$
⟨*proof*⟩

lemma *sep-conj-empty'*[*simp*]:
$(\square ** P) = P$
⟨*proof*⟩

lemma *sep-conj-sep-emptyI*:
$P\ s \Longrightarrow (P ** \square)\ s$
⟨*proof*⟩

lemma *sep-conj-true-P*[*simp*]:
$(sep\text{-}true ** (sep\text{-}true ** P)) = (sep\text{-}true ** P)$
⟨*proof*⟩

lemma *sep-conj-disj*:
$((\lambda s.\ P\ s \vee Q\ s) ** R)\ s = ((P ** R)\ s \vee (Q ** R)\ s)$ (**is** $?x = (?y \vee ?z)$)
⟨*proof*⟩

lemma *sep-conj-conj*:
$((\lambda s.\ P\ s \wedge Q\ s) ** R)\ s \Longrightarrow (P ** R)\ s \wedge (Q ** R)\ s$
⟨*proof*⟩

lemma *sep-conj-exists1*:
$((\lambda s.\ \exists x.\ P\ x\ s) ** Q)\ s = (\exists x.\ (P\ x ** Q)\ s)$
⟨*proof*⟩

lemma *sep-conj-exists2*:
$(P ** (\lambda s.\ \exists x.\ Q\ x\ s)) = (\lambda s.\ (\exists x.\ (P ** Q\ x)\ s))$
⟨*proof*⟩

lemmas *sep-conj-exists* = *sep-conj-exists1 sep-conj-exists2*

lemma *sep-conj-forall*:
$((\lambda s.\ \forall x.\ P\ x\ s) ** Q)\ s \Longrightarrow (P\ x ** Q)\ s$
⟨*proof*⟩

lemma *sep-conj-impl*:
$[\![(P ** Q)\ s; \bigwedge s.\ P\ s \Longrightarrow P'\ s; \bigwedge s.\ Q\ s \Longrightarrow Q'\ s]\!] \Longrightarrow (P' ** Q')\ s$
⟨*proof*⟩

lemma *sep-conj-impl1*:
assumes *P*: $\bigwedge s.\ P\ s \Longrightarrow I\ s$
shows $(P ** R)\ s \Longrightarrow (I ** R)\ s$
⟨*proof*⟩

lemma *sep-conj-sep-true-left*:
$(P ** Q)\ s \Longrightarrow (\textit{sep-true} ** Q)\ s$
⟨*proof*⟩

lemma *sep-conj-sep-true-right*:
$(P ** Q)\ s \Longrightarrow (P ** \textit{sep-true})\ s$
⟨*proof*⟩

lemma *sep-globalise*:
$[\![(P ** R)\ s; (\bigwedge s.\ P\ s \Longrightarrow Q\ s)]\!] \Longrightarrow (Q ** R)\ s$
⟨*proof*⟩

lemma *sep-implI*:
assumes *a*: $\bigwedge h'\ h''.\ [\![h \oplus h' = \textit{Some}\ h''; P\ h']\!] \Longrightarrow Q\ h''$
shows $(P \longrightarrow^* Q)\ h$
⟨*proof*⟩

lemma *sep-implD*:
$(x \longrightarrow^* y)\ h \Longrightarrow \forall h'\ h''.\ h \oplus h' = \textit{Some}\ h'' \wedge x\ h' \longrightarrow y\ h''$
⟨*proof*⟩

lemma *sep-impl-sep-true*[*simp*]:
$(P \longrightarrow^* \textit{sep-true}) = \textit{sep-true}$
⟨*proof*⟩

lemma *sep-impl-sep-false*[*simp*]:
$(\textit{sep-false} \longrightarrow^* P) = \textit{sep-true}$
⟨*proof*⟩

lemma *sep-impl-sep-true-P*:
$(\textit{sep-true} \longrightarrow^* P)\ s \Longrightarrow P\ s$
⟨*proof*⟩

lemma *sep-impl-sep-true-false*[*simp*]:
$(\textit{sep-true} \longrightarrow^* \textit{sep-false}) = \textit{sep-false}$
⟨*proof*⟩

lemma *sep-conj-sep-impl*:
$[\![P\ s; \bigwedge s.\ (P ** Q)\ s \Longrightarrow R\ s]\!] \Longrightarrow (Q \longrightarrow^* R)\ s$
⟨*proof*⟩

lemma *sep-conj-sep-impl2*:
$[\![(P ** Q)\ s; \bigwedge s.\ P\ s \Longrightarrow (Q \longrightarrow^* R)\ s]\!] \Longrightarrow R\ s$
⟨*proof*⟩

lemma *sep-conj-sep-impl-sep-conj2*:
$(P ** R)\ s \Longrightarrow (P ** (Q \longrightarrow^* (Q ** R)))\ s$
⟨*proof*⟩

lemma *sep-conj-triv-strip2*:
$Q = R \Longrightarrow (Q ** P) = (R ** P)$ ⟨*proof*⟩

end

end

23 Equivalence between Separation Algebra Formulations

theory *Sep-Eq*
imports *Separation-Algebra Separation-Algebra-Alt*
begin

In this theory we show that our total formulation of separation algebra is equivalent in strength to Calcagno et al's original partial one.

This theory is not intended to be included in own developments.

no-notation *map-add* (**infixl** ‹++› *100*)

24 Total implies Partial

definition *add2* :: *'a::sep-algebra => 'a => 'a option* **where**
add2 x y ≡ if x ## y then Some (x + y) else None

lemma *add2-zero*: *add2 x 0 = Some x*
⟨*proof*⟩

lemma *add2-comm*: *add2 x y = add2 y x*
⟨*proof*⟩

lemma *add2-assoc*:
lift2 add2 a (lift2 add2 b c) = lift2 add2 (lift2 add2 a b) c
⟨*proof*⟩

interpretation *total-partial*: *sep-algebra-alt 0 add2*
⟨*proof*⟩

25 Partial implies Total

definition
sep-add' :: *'a ⇒ 'a ⇒ 'a* :: *sep-algebra-alt* **where**
sep-add' x y ≡ if disjoint x y then the (add x y) else undefined

lemma *sep-disj-zero'*:
disjoint x 0
⟨*proof*⟩

lemma *sep-disj-commuteI'*:
disjoint x y ⟹ disjoint y x
⟨*proof*⟩

lemma *sep-add-zero'*:
sep-add' x 0 = x
⟨*proof*⟩

lemma *sep-add-commute'*:
disjoint x y ⟹ sep-add' x y = sep-add' y x
⟨*proof*⟩

lemma *sep-add-assoc'*:
⟦ *disjoint x y*; *disjoint y z*; *disjoint x z* ⟧ ⟹
sep-add' (sep-add' x y) z = sep-add' x (sep-add' y z)
⟨*proof*⟩

lemma *sep-disj-addD1'*:
disjoint x (sep-add' y z) ⟹ disjoint y z ⟹ disjoint x y
⟨*proof*⟩

lemma *sep-disj-addI1'*:
disjoint x (*sep-add' y z*) $\Longrightarrow$ *disjoint y z* $\Longrightarrow$ *disjoint* (*sep-add' x y*) *z*
⟨*proof*⟩

interpretation *partial-total*: *sep-algebra sep-add' 0 disjoint*
⟨*proof*⟩

end

26 A simplified version of the actual capDL specification.

theory *Types-D*
imports *HOL−Library.Word*
begin

type-synonym *cdl-object-id* = *32 word*

type-synonym *cdl-object-set* = *cdl-object-id set*

type-synonym *cdl-size-bits* = *nat*

type-synonym *cdl-cnode-index* = *nat*

type-synonym *cdl-cap-ref* = *cdl-object-id* × *cdl-cnode-index*

datatype *cdl-right* = *AllowRead* | *AllowWrite* | *AllowGrant*

datatype *cdl-cap* =
NullCap
| *EndpointCap cdl-object-id cdl-right set*
| *CNodeCap cdl-object-id*
| *TcbCap cdl-object-id*

type-synonym *cdl-cap-map* = *cdl-cnode-index* ⇒ *cdl-cap option*

translations
(*type*) *cdl-cap-map* <= (*type*) *nat* ⇒ *cdl-cap option*
(*type*) *cdl-cap-ref* <= (*type*) *cdl-object-id* × *nat*

```
type-synonym cdl-cptr = 32 word

record cdl-tcb =
  cdl-tcb-caps :: cdl-cap-map
  cdl-tcb-fault-endpoint :: cdl-cptr

record cdl-cnode =
  cdl-cnode-caps :: cdl-cap-map
  cdl-cnode-size-bits :: cdl-size-bits

datatype cdl-object =
    Endpoint
  | Tcb cdl-tcb
  | CNode cdl-cnode

type-synonym cdl-heap = cdl-object-id ⇒ cdl-object option
type-synonym cdl-component  = nat option
type-synonym cdl-components = cdl-component set
type-synonym cdl-ghost-state = cdl-object-id ⇒ cdl-components

translations
  (type) cdl-heap <= (type) cdl-object-id ⇒ cdl-object option
  (type) cdl-ghost-state <= (type) cdl-object-id ⇒ nat option set

record cdl-state =
  cdl-objects :: cdl-heap
  cdl-current-thread :: cdl-object-id option
  cdl-ghost-state :: cdl-ghost-state

datatype cdl-object-type =
    EndpointType
  | TcbType
  | CNodeType

definition
  object-type :: cdl-object ⇒ cdl-object-type
where
  object-type x ≡
    case x of
```

Endpoint ⇒ *EndpointType*
| *Tcb* - ⇒ *TcbType*
| *CNode* - ⇒ *CNodeType*

definition *cap-objects* :: *cdl-cap* ⇒ *cdl-object-id set*
where
cap-objects cap ≡
case cap of
TcbCap x ⇒ {*x*}
| *CNodeCap x* ⇒ {*x*}
| *EndpointCap x* - ⇒ {*x*}

definition *cap-has-object* :: *cdl-cap* ⇒ *bool*
where
cap-has-object cap ≡
case cap of
NullCap ⇒ *False*
| - ⇒ *True*

definition *cap-object* :: *cdl-cap* ⇒ *cdl-object-id*
where
cap-object cap ≡
if cap-has-object cap
then THE obj-id. cap-objects cap = {*obj-id*}
else undefined

lemma *cap-object-simps*:
cap-object (*TcbCap x*) = *x*
cap-object (*CNodeCap x*) = *x*
cap-object (*EndpointCap x j*) = *x*
⟨*proof*⟩

definition
cap-rights :: *cdl-cap* ⇒ *cdl-right set*
where
cap-rights c ≡ *case c of*
EndpointCap - x ⇒ *x*
| - ⇒ *UNIV*

definition
update-cap-rights :: *cdl-right set* ⇒ *cdl-cap* ⇒ *cdl-cap*
where
update-cap-rights r c ≡ *case c of*
EndpointCap f1 - ⇒ *EndpointCap f1 r*
| - ⇒ *c*

definition
object-slots :: *cdl-object* ⇒ *cdl-cap-map*
where
object-slots obj ≡ *case obj of*
  CNode x ⇒ *cdl-cnode-caps x*
| *Tcb x* ⇒ *cdl-tcb-caps x*
| - ⇒ *Map.empty*

definition
update-slots :: *cdl-cap-map* ⇒ *cdl-object* ⇒ *cdl-object*
where
update-slots new-val obj ≡ *case obj of*
  CNode x ⇒ *CNode* (*x*(|*cdl-cnode-caps* := *new-val*|))
| *Tcb x* ⇒ *Tcb* (*x*(|*cdl-tcb-caps* := *new-val*|))
| - ⇒ *obj*

definition
add-to-slots :: *cdl-cap-map* ⇒ *cdl-object* ⇒ *cdl-object*
where
add-to-slots new-val obj ≡ *update-slots* (*new-val* ++ (*object-slots obj*)) *obj*

definition
slots-of :: *cdl-heap* ⇒ *cdl-object-id* ⇒ *cdl-cap-map*
where
slots-of h ≡ λ*obj-id*.
case h obj-id of
  None ⇒ *Map.empty*
| *Some obj* ⇒ *object-slots obj*

definition
has-slots :: *cdl-object* ⇒ *bool*
where
has-slots obj ≡ *case obj of*
  CNode - ⇒ *True*
| *Tcb* - ⇒ *True*
| - ⇒ *False*

definition
object-at :: (*cdl-object* ⇒ *bool*) ⇒ *cdl-object-id* ⇒ *cdl-heap* ⇒ *bool*
where
object-at P p s ≡ ∃ *object*. *s p* = *Some object* ∧ *P object*

abbreviation
ko-at k ≡ *object-at* ((=) *k*)

end

27 Instantiating capDL as a separation algebra.

theory *Abstract-Separation-D*
imports *../../Sep-Tactics Types-D ../../Map-Extra*
begin

lemma *inter-empty-not-both*:
$[\![x \in A; A \cap B = \{\}]\!] \Longrightarrow x \notin B$
⟨*proof*⟩

lemma *union-intersection*:
$A \cap (A \cup B) = A$
$B \cap (A \cup B) = B$
$(A \cup B) \cap A = A$
$(A \cup B) \cap B = B$
⟨*proof*⟩

lemma *union-intersection1*: $A \cap (A \cup B) = A$
⟨*proof*⟩
lemma *union-intersection2*: $B \cap (A \cup B) = B$
⟨*proof*⟩

lemma *restrict-map-disj'*:
$S \cap T = \{\} \Longrightarrow h\ |`\ S \perp h'\ |`\ T$
⟨*proof*⟩

lemma *map-add-restrict-comm*:
$S \cap T = \{\} \Longrightarrow h\ |`\ S\ ++\ h'\ |`\ T = h'\ |`\ T\ ++\ h\ |`\ S$
⟨*proof*⟩

datatype *sep-state* = *SepState cdl-heap cdl-ghost-state*

primrec *sep-heap* :: *sep-state* ⇒ *cdl-heap*
where *sep-heap* (*SepState h gs*) = *h*

primrec *sep-ghost-state* :: *sep-state* ⇒ *cdl-ghost-state*
where *sep-ghost-state* (*SepState h gs*) = *gs*

definition
the-set :: *'a option set* ⇒ *'a set*
where
the-set xs = {*x*. *Some x* ∈ *xs*}

lemma *the-set-union* [*simp*]:
the-set (*A* ∪ *B*) = *the-set A* ∪ *the-set B*
⟨*proof*⟩

lemma *the-set-inter* [*simp*]:
the-set (*A* ∩ *B*) = *the-set A* ∩ *the-set B*
⟨*proof*⟩

lemma *the-set-inter-empty*:
A ∩ *B* = {} ⟹ *the-set A* ∩ *the-set B* = {}
⟨*proof*⟩

definition
slots-of-heap :: *cdl-heap* ⇒ *cdl-object-id* ⇒ *cdl-cap-map*
where
slots-of-heap h ≡ λ*obj-id*.
case h obj-id of
  None ⇒ *Map.empty*
| *Some obj* ⇒ *object-slots obj*

definition
add-to-slots :: *cdl-cap-map* ⇒ *cdl-object* ⇒ *cdl-object*
where
add-to-slots new-val obj ≡ *update-slots* (*new-val* ++ (*object-slots obj*)) *obj*

lemma *add-to-slots-assoc*:
add-to-slots x (*add-to-slots* (*y* ++ *z*) *obj*) =
 add-to-slots (*x* ++ *y*) (*add-to-slots z obj*)
⟨*proof*⟩

lemma *add-to-slots-twice* [*simp*]:
add-to-slots x (*add-to-slots y a*) = *add-to-slots* (*x* ++ *y*) *a*
⟨*proof*⟩

lemma *slots-of-heap-empty* [*simp*]: *slots-of-heap Map.empty object-id* = *Map.empty*
⟨*proof*⟩

lemma *slots-of-heap-empty2* [*simp*]:
h obj-id = *None* ⟹ *slots-of-heap h obj-id* = *Map.empty*

⟨*proof*⟩

lemma *update-slots-add-to-slots-empty* [*simp*]:
update-slots Map.empty (*add-to-slots new obj*) = *update-slots Map.empty obj*
⟨*proof*⟩

lemma *update-object-slots-id* [*simp*]: *update-slots* (*object-slots a*) *a* = *a*
⟨*proof*⟩

lemma *update-slots-of-heap-id* [*simp*]:
h obj-id = *Some obj* ⟹ *update-slots* (*slots-of-heap h obj-id*) *obj* = *obj*
⟨*proof*⟩

lemma *add-to-slots-empty* [*simp*]: *add-to-slots Map.empty h* = *h*
⟨*proof*⟩

lemma *update-slots-eq*:
update-slots a o1 = *update-slots a o2* ⟹ *update-slots b o1* = *update-slots b o2*
⟨*proof*⟩

definition
not-conflicting-objects :: *sep-state* ⇒ *sep-state* ⇒ *cdl-object-id* ⇒ *bool*
where
not-conflicting-objects state-a state-b = (*λobj-id.*
let heap-a = *sep-heap state-a*;
heap-b = *sep-heap state-b*;
gs-a = *sep-ghost-state state-a*;
gs-b = *sep-ghost-state state-b*
in case (*heap-a obj-id*, *heap-b obj-id*) *of*
(*Some o1*, *Some o2*) ⇒ *object-type o1* = *object-type o2* ∧ *gs-a obj-id* ∩ *gs-b obj-id* = {}
| - ⇒ *True*)

definition
clean-slots :: *cdl-cap-map* ⇒ *cdl-components* ⇒ *cdl-cap-map*
where
clean-slots slots cmp ≡ *slots* |‘ *the-set cmp*

definition
object-clean-fields :: *cdl-object* ⇒ *cdl-components* ⇒ *cdl-object*
where
object-clean-fields obj cmp ≡ *if None* ∈ *cmp then obj else case obj of*
Tcb x ⇒ *Tcb* (*x*⦇*cdl-tcb-fault-endpoint* := *undefined*⦈)

| *CNode x* ⇒ *CNode* (*x*⦇*cdl-cnode-size-bits* := *undefined* ⦈)
| - ⇒ *obj*

definition
object-clean-slots :: *cdl-object* ⇒ *cdl-components* ⇒ *cdl-object*
where
object-clean-slots obj cmp ≡ *update-slots* (*clean-slots* (*object-slots obj*) *cmp*) *obj*

definition
object-clean :: *cdl-object* ⇒ *cdl-components* ⇒ *cdl-object*
where
object-clean obj gs ≡ *object-clean-slots* (*object-clean-fields obj gs*) *gs*

definition
object-add :: *cdl-object* ⇒ *cdl-object* ⇒ *cdl-components* ⇒ *cdl-components* ⇒ *cdl-object*
where
object-add obj-a obj-b cmps-a cmps-b ≡
let clean-obj-a = *object-clean obj-a cmps-a*;
clean-obj-b = *object-clean obj-b cmps-b*
in if (*cmps-a* = {})
then clean-obj-b
else if (*cmps-b* = {})
then clean-obj-a
else if (*None* ∈ *cmps-b*)
then (*update-slots* (*object-slots clean-obj-a* ++ *object-slots clean-obj-b*) *clean-obj-b*)
else (*update-slots* (*object-slots clean-obj-a* ++ *object-slots clean-obj-b*) *clean-obj-a*)

definition
cdl-heap-add :: *sep-state* ⇒ *sep-state* ⇒ *cdl-heap*
where
cdl-heap-add state-a state-b ≡ λ*obj-id*.
let heap-a = *sep-heap state-a*;
heap-b = *sep-heap state-b*;
gs-a = *sep-ghost-state state-a*;
gs-b = *sep-ghost-state state-b*
in case heap-b obj-id of
None ⇒ *heap-a obj-id*
| *Some obj-b* ⇒ *case heap-a obj-id of*
None ⇒ *heap-b obj-id*
| *Some obj-a* ⇒ *Some* (*object-add obj-a obj-b* (*gs-a obj-id*) (*gs-b obj-id*))

definition

cdl-ghost-state-add :: *sep-state* ⇒ *sep-state* ⇒ *cdl-ghost-state*
where
cdl-ghost-state-add state-a state-b ≡ *λobj-id.*
let heap-a = *sep-heap state-a*;
heap-b = *sep-heap state-b*;
gs-a = *sep-ghost-state state-a*;
gs-b = *sep-ghost-state state-b*
in *if heap-a obj-id* = *None* ∧ *heap-b obj-id* ≠ *None then gs-b obj-id*
else if heap-b obj-id = *None* ∧ *heap-a obj-id* ≠ *None then gs-a obj-id*
else gs-a obj-id ∪ *gs-b obj-id*

definition
sep-state-add :: *sep-state* ⇒ *sep-state* ⇒ *sep-state*
where
sep-state-add state-a state-b ≡
let
heap-a = *sep-heap state-a*;
heap-b = *sep-heap state-b*;
gs-a = *sep-ghost-state state-a*;
gs-b = *sep-ghost-state state-b*
in
SepState (*cdl-heap-add state-a state-b*) (*cdl-ghost-state-add state-a state-b*)

definition
sep-state-disj :: *sep-state* ⇒ *sep-state* ⇒ *bool*
where
sep-state-disj state-a state-b ≡
let
heap-a = *sep-heap state-a*;
heap-b = *sep-heap state-b*;
gs-a = *sep-ghost-state state-a*;
gs-b = *sep-ghost-state state-b*
in
∀ *obj-id. not-conflicting-objects state-a state-b obj-id*

lemma *not-conflicting-objects-comm*:
not-conflicting-objects h1 h2 obj = *not-conflicting-objects h2 h1 obj*
⟨*proof*⟩

lemma *object-clean-comm*:
⟦*object-type obj-a* = *object-type obj-b*;
object-slots obj-a ++ *object-slots obj-b* = *object-slots obj-b* ++ *object-slots obj-a*;
None ∉ *cmp*⟧
⟹ *object-clean* (*add-to-slots* (*object-slots obj-a*) *obj-b*) *cmp* =
object-clean (*add-to-slots* (*object-slots obj-b*) *obj-a*) *cmp*

⟨*proof*⟩

lemma *add-to-slots-object-slots*:
object-type y = *object-type z*
⟹ *add-to-slots* (*object-slots* (*add-to-slots* (*x*) *y*)) *z* =
add-to-slots (*x* ++ *object-slots y*) *z*
⟨*proof*⟩

lemma *not-conflicting-objects-empty* [*simp*]:
not-conflicting-objects s (*SepState Map.empty* (λ*obj-id*. {})) *obj-id*
⟨*proof*⟩

lemma *empty-not-conflicting-objects* [*simp*]:
not-conflicting-objects (*SepState Map.empty* (λ*obj-id*. {})) *s obj-id*
⟨*proof*⟩

lemma *not-conflicting-objects-empty-object* [*elim*!]:
(*sep-heap x*) *obj-id* = *None* ⟹ *not-conflicting-objects x y obj-id*
⟨*proof*⟩

lemma *empty-object-not-conflicting-objects* [*elim*!]:
(*sep-heap y*) *obj-id* = *None* ⟹ *not-conflicting-objects x y obj-id*
⟨*proof*⟩

lemma *cdl-heap-add-empty* [*simp*]:
cdl-heap-add (*SepState h gs*) (*SepState Map.empty* (λ*obj-id*. {})) = *h*
⟨*proof*⟩

lemma *empty-cdl-heap-add* [*simp*]:
cdl-heap-add (*SepState Map.empty* (λ*obj-id*. {})) (*SepState h gs*)= *h*
⟨*proof*⟩

lemma *map-add-result-empty1*: *a* ++ *b* = *Map.empty* ⟹ *a* = *Map.empty*
⟨*proof*⟩

lemma *map-add-result-empty2*: *a* ++ *b* = *Map.empty* ⟹ *b* = *Map.empty*
⟨*proof*⟩

lemma *map-add-emptyE* [*elim*!]: ⟦*a* ++ *b* = *Map.empty*; ⟦*a* = *Map.empty*; *b* = *Map.empty*⟧ ⟹ *R*⟧ ⟹ *R*
⟨*proof*⟩

lemma *clean-slots-empty* [*simp*]:
clean-slots Map.empty cmp = *Map.empty*
⟨*proof*⟩

lemma *object-type-update-slots* [*simp*]:
object-type (*update-slots slots x*) = *object-type x*
⟨*proof*⟩

lemma *object-type-object-clean-slots* [*simp*]:
object-type (*object-clean-slots x cmp*) = *object-type x*
⟨*proof*⟩

lemma *object-type-object-clean-fields* [*simp*]:
object-type (*object-clean-fields x cmp*) = *object-type x*
⟨*proof*⟩

lemma *object-type-object-clean* [*simp*]:
object-type (*object-clean x cmp*) = *object-type x*
⟨*proof*⟩

lemma *object-type-add-to-slots* [*simp*]:
object-type (*add-to-slots slots x*) = *object-type x*
⟨*proof*⟩

lemma *object-slots-update-slots* [*simp*]:
has-slots obj ⟹ *object-slots* (*update-slots slots obj*) = *slots*
⟨*proof*⟩

lemma *object-slots-update-slots-empty* [*simp*]:
¬*has-slots obj* ⟹ *object-slots* (*update-slots slots obj*) = *Map.empty*
⟨*proof*⟩

lemma *update-slots-no-slots* [*simp*]:
¬*has-slots obj* ⟹ *update-slots slots obj* = *obj*
⟨*proof*⟩

lemma *update-slots-update-slots* [*simp*]:
update-slots slots (*update-slots slots′ obj*) = *update-slots slots obj*
⟨*proof*⟩

lemma *update-slots-same-object*:
a = *b* ⟹ *update-slots a obj* = *update-slots b obj*
⟨*proof*⟩

lemma *object-type-has-slots*:
⟦*has-slots x*; *object-type x* = *object-type y*⟧ ⟹ *has-slots y*
⟨*proof*⟩

lemma *object-slots-object-clean-fields* [*simp*]:
object-slots (*object-clean-fields obj cmp*) = *object-slots obj*
⟨*proof*⟩

lemma *object-slots-object-clean-slots* [*simp*]:
object-slots (*object-clean-slots obj cmp*) = *clean-slots* (*object-slots obj*) *cmp*
⟨*proof*⟩

lemma *object-slots-object-clean* [*simp*]:
object-slots (*object-clean obj cmp*) = *clean-slots* (*object-slots obj*) *cmp*
⟨*proof*⟩

lemma *object-slots-add-to-slots* [*simp*]:
object-type y = *object-type z* ⟹ *object-slots* (*add-to-slots* (*object-slots y*) *z*) = *object-slots y* ++ *object-slots z*
⟨*proof*⟩

lemma *update-slots-object-clean-slots* [*simp*]:
update-slots slots (*object-clean-slots obj cmp*) = *update-slots slots obj*
⟨*proof*⟩

lemma *object-clean-fields-idem* [*simp*]:
object-clean-fields (*object-clean-fields obj cmp*) *cmp* = *object-clean-fields obj cmp*
⟨*proof*⟩

lemma *object-clean-slots-idem* [*simp*]:
object-clean-slots (*object-clean-slots obj cmp*) *cmp* = *object-clean-slots obj cmp*
⟨*proof*⟩

lemma *object-clean-fields-object-clean-slots* [*simp*]:
object-clean-fields (*object-clean-slots obj gs*) *gs* = *object-clean-slots* (*object-clean-fields obj gs*) *gs*
⟨*proof*⟩

lemma *object-clean-idem* [*simp*]:
object-clean (*object-clean obj cmp*) *cmp* = *object-clean obj cmp*
⟨*proof*⟩

lemma *has-slots-object-clean-slots*:
has-slots (*object-clean-slots obj cmp*) = *has-slots obj*
⟨*proof*⟩

lemma *has-slots-object-clean-fields*:
has-slots (*object-clean-fields obj cmp*) = *has-slots obj*
⟨*proof*⟩

lemma *has-slots-object-clean*:
has-slots (*object-clean obj cmp*) = *has-slots obj*
⟨*proof*⟩

lemma *object-slots-update-slots-object-clean-fields* [*simp*]:
object-slots (*update-slots slots* (*object-clean-fields obj cmp*)) = *object-slots* (*update-slots slots obj*)
⟨*proof*⟩

lemma *object-clean-fields-update-slots* [*simp*]:
object-clean-fields (*update-slots slots obj*) *cmp* = *update-slots slots* (*object-clean-fields*

obj cmp)
⟨*proof*⟩

lemma *object-clean-fields-twice* [*simp*]:
(*object-clean-fields* (*object-clean-fields obj cmp′*) *cmp*) = *object-clean-fields obj* (*cmp* ∩ *cmp′*)
⟨*proof*⟩

lemma *update-slots-object-clean-fields*:
⟦*None* ∉ *cmps*; *None* ∉ *cmps′*; *object-type obj* = *object-type obj′*⟧
⟹ *update-slots slots* (*object-clean-fields obj cmps*) =
update-slots slots (*object-clean-fields obj′ cmps′*)
⟨*proof*⟩

lemma *object-clean-fields-no-slots*:
⟦*None* ∉ *cmps*; *None* ∉ *cmps′*; *object-type obj* = *object-type obj′*; ¬ *has-slots obj*; ¬ *has-slots obj′*⟧
⟹ *object-clean-fields obj cmps* = *object-clean-fields obj′ cmps′*
⟨*proof*⟩

lemma *update-slots-object-clean*:
⟦*None* ∉ *cmps*; *None* ∉ *cmps′*; *object-type obj* = *object-type obj′*⟧
⟹ *update-slots slots* (*object-clean obj cmps*) = *update-slots slots* (*object-clean obj′ cmps′*)
⟨*proof*⟩

lemma *cdl-heap-add-assoc′*:
∀ *obj-id*. *not-conflicting-objects x z obj-id* ∧
not-conflicting-objects y z obj-id ∧
not-conflicting-objects x z obj-id ⟹
cdl-heap-add (*SepState* (*cdl-heap-add x y*) (*cdl-ghost-state-add x y*)) *z* =
cdl-heap-add x (*SepState* (*cdl-heap-add y z*) (*cdl-ghost-state-add y z*))
⟨*proof*⟩

lemma *cdl-heap-add-assoc*:
⟦*sep-state-disj x y*; *sep-state-disj y z*; *sep-state-disj x z*⟧
⟹ *cdl-heap-add* (*SepState* (*cdl-heap-add x y*) (*cdl-ghost-state-add x y*)) *z* =
cdl-heap-add x (*SepState* (*cdl-heap-add y z*) (*cdl-ghost-state-add y z*))
⟨*proof*⟩

lemma *cdl-ghost-state-add-assoc*:
cdl-ghost-state-add (*SepState* (*cdl-heap-add x y*) (*cdl-ghost-state-add x y*)) *z* =
cdl-ghost-state-add x (*SepState* (*cdl-heap-add y z*) (*cdl-ghost-state-add y z*))
⟨*proof*⟩

lemma *clean-slots-map-add-comm*:
cmps-a ∩ *cmps-b* = {}
⟹ *clean-slots slots-a cmps-a* ++ *clean-slots slots-b cmps-b* =
clean-slots slots-b cmps-b ++ *clean-slots slots-a cmps-a*

⟨*proof*⟩

lemma *object-clean-all*:
object-type obj-a = *object-type obj-b* ⟹ *object-clean obj-b* {} = *object-clean obj-a* {}
⟨*proof*⟩

lemma *object-add-comm*:
⟦*object-type obj-a* = *object-type obj-b*; *cmps-a* ∩ *cmps-b* = {}⟧
⟹ *object-add obj-a obj-b cmps-a cmps-b* = *object-add obj-b obj-a cmps-b cmps-a*
⟨*proof*⟩

lemma *sep-state-add-comm*:
sep-state-disj x y ⟹ *sep-state-add x y* = *sep-state-add y x*
⟨*proof*⟩

lemma *add-to-slots-comm*:
⟦*object-slots y-obj* ⊥ *object-slots z-obj*; *update-slots Map.empty y-obj* = *update-slots Map.empty z-obj* ⟧
⟹ *add-to-slots* (*object-slots z-obj*) *y-obj* = *add-to-slots* (*object-slots y-obj*) *z-obj*
⟨*proof*⟩

lemma *cdl-heap-add-none1*:
cdl-heap-add x y obj-id = *None* ⟹ (*sep-heap x*) *obj-id* = *None*
⟨*proof*⟩

lemma *cdl-heap-add-none2*:
cdl-heap-add x y obj-id = *None* ⟹ (*sep-heap y*) *obj-id* = *None*
⟨*proof*⟩

lemma *object-type-object-addL*:
object-type obj = *object-type obj'*
⟹ *object-type* (*object-add obj obj' cmp cmp'*) = *object-type obj*
⟨*proof*⟩

lemma *object-type-object-addR*:
object-type obj = *object-type obj'*
⟹ *object-type* (*object-add obj obj' cmp cmp'*) = *object-type obj'*
⟨*proof*⟩

lemma *sep-state-add-disjL*:
⟦*sep-state-disj y z*; *sep-state-disj x* (*sep-state-add y z*)⟧ ⟹ *sep-state-disj x y*
⟨*proof*⟩

lemma *sep-state-add-disjR*:
⟦*sep-state-disj y z*; *sep-state-disj x* (*sep-state-add y z*)⟧ ⟹ *sep-state-disj x z*
⟨*proof*⟩

lemma *sep-state-add-disj*:

⟦*sep-state-disj y z*; *sep-state-disj x y*; *sep-state-disj x z*⟧ ⟹ *sep-state-disj x* (*sep-state-add y z*)
⟨*proof*⟩

instantiation *sep-state* :: *zero*
begin
definition *0* ≡ *SepState Map.empty* (λ*obj-id*. {})
instance ⟨*proof*⟩
end

instantiation *sep-state* :: *stronger-sep-algebra*
begin

definition (##) ≡ *sep-state-disj*
definition (+) ≡ *sep-state-add*

instance
⟨*proof*⟩

end

end

28 Defining some separation logic maps-to predicates on top of the instantiation.

theory *Separation-D*
imports *Abstract-Separation-D*
begin

type-synonym *sep-pred* = *sep-state* ⇒ *bool*

definition
state-sep-projection :: *cdl-state* ⇒ *sep-state*
where
state-sep-projection ≡ λ*s*. *SepState* (*cdl-objects s*) (*cdl-ghost-state s*)

abbreviation
lift′ :: (*sep-state* ⇒ ′*a*) ⇒ *cdl-state* ⇒ ′*a* (‹<->›)
where
<*P*> *s* ≡ *P* (*state-sep-projection s*)

definition
sep-map-general :: *cdl-object-id* ⇒ *cdl-object* ⇒ *cdl-components* ⇒ *sep-pred*
where
sep-map-general p obj gs ≡ λ*s*. *sep-heap s* = [*p* ↦ *obj*] ∧ *sep-ghost-state s p* = *gs*

lemma *sep-map-general-def2*:
sep-map-general p obj gs s =
(*dom* (*sep-heap s*) = {*p*} ∧ *ko-at obj p* (*sep-heap s*) ∧ *sep-ghost-state s p* = *gs*)
⟨*proof*⟩

definition
sep-map-i :: *cdl-object-id* ⇒ *cdl-object* ⇒ *sep-pred* (‹- ↦i -› [76,71] 76)
where
p ↦*i obj* ≡ *sep-map-general p obj UNIV*

definition
sep-map-f :: *cdl-object-id* ⇒ *cdl-object* ⇒ *sep-pred* (‹- ↦f -› [76,71] 76)
where
p ↦*f obj* ≡ *sep-map-general p* (*update-slots Map.empty obj*) {*None*}

definition
sep-map-c :: *cdl-cap-ref* ⇒ *cdl-cap* ⇒ *sep-pred* (‹- ↦c -› [76,71] 76)
where
p ↦*c cap* ≡ λ*s*. *let* (*obj-id*, *slot*) = *p*; *heap* = *sep-heap s in*
∃ *obj*. *sep-map-general obj-id obj* {*Some slot*} *s* ∧ *object-slots obj* = [*slot* ↦ *cap*]

definition
sep-any :: (′*a* ⇒ ′*b* ⇒ *sep-pred*) ⇒ (′*a* ⇒ *sep-pred*) **where**
sep-any m ≡ (λ*p s*. ∃ *v*. (*m p v*) *s*)

abbreviation *sep-any-map-i* ≡ *sep-any sep-map-i*
notation *sep-any-map-i* (‹- ↦i −› 76)

abbreviation *sep-any-map-c* ≡ *sep-any sep-map-c*
notation *sep-any-map-c* (‹- ↦c −› 76)

end

References

[1] G. Klein, R. Kolanski, and A. Boyton. Mechanised separation algebra (rough diamond). In Beringer and Felty, editors, *Interactive Theorem Proving (ITP 2012)*, LNCS. Springer, 2012.