# 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* :: $'b \Rightarrow 'a \Rightarrow 'b$ ($\langle\langle$-$\rangle\rangle$) **where**
$\langle f \rangle \equiv \lambda s.\ f$

**abbreviation** (*input*)
*pred-ex* :: $('b \Rightarrow 'a \Rightarrow bool) \Rightarrow 'a \Rightarrow bool$ (**binder** $\langle EXS\ \rangle\ 10$) **where**
$EXS\ x.\ P\ x \equiv \lambda s.\ \exists x.\ P\ x\ s$

**abbreviation** (*input*)
*pred-all* :: $('b \Rightarrow 'a \Rightarrow bool) \Rightarrow 'a \Rightarrow bool$ (**binder** $\langle ALLS\ \rangle\ 10$) **where**
$ALLS\ x.\ P\ x \equiv \lambda s.\ \forall x.\ P\ x\ s$

# 3 Associative/Commutative Monoid Basis of Separation Algebras

**class** *pre-sep-algebra* = *zero* + *plus* +
**fixes** *sep-disj* :: $'a \Rightarrow 'a \Rightarrow bool$ (**infix** $\langle\#\#\rangle\ 60$)

**assumes** *sep-disj-zero* [*simp*]: $x\ \#\#\ 0$
**assumes** *sep-disj-commuteI*: $x\ \#\#\ y \Longrightarrow y\ \#\#\ x$

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

**assumes** *sep-add-assoc*:
$[\![\ x\ \#\#\ y;\ y\ \#\#\ z;\ x\ \#\#\ z\ ]\!] \Longrightarrow (x + y) + z = x + (y + z)$
**begin**

**lemma** *sep-disj-commute*: $x\ \#\#\ y = y\ \#\#\ x$
**by** (*blast intro*: *sep-disj-commuteI*)

**lemma** *sep-add-left-commute*:
**assumes** *a*: $a\ \#\#\ b\ b\ \#\#\ c\ a\ \#\#\ c$
**shows** $b + (a + c) = a + (b + c)$ (**is** *?lhs* = *?rhs*)
**proof** −
**have** *?lhs* = $b + a + c$ **using** *a*
**by** (*simp add*: *sep-add-assoc*[*symmetric*] *sep-disj-commute*)
**also have** ... = $a + b + c$ **using** *a*
**by** (*simp add*: *sep-add-commute sep-disj-commute*)
**also have** ... = *?rhs* **using** *a*
**by** (*simp add*: *sep-add-assoc sep-disj-commute*)
**finally show** *?thesis* **.**
**qed**

**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*: $[\![ x \mathbin{\#\#} y + z;\ y \mathbin{\#\#} z ]\!] \Longrightarrow x \mathbin{\#\#} y$
  **assumes** *sep-disj-addI1*: $[\![ x \mathbin{\#\#} y + z;\ y \mathbin{\#\#} z ]\!] \Longrightarrow x + y \mathbin{\#\#} z$
**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* $\equiv \langle True \rangle$

**abbreviation**
  *sep-false* $\equiv \langle False \rangle$

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

## 4.2 Disjunction/Addition Properties

**lemma** *disjoint-zero-sym* [*simp*]: $0 \mathbin{\#\#} x$
  **by** (*simp add*: *sep-disj-commute*)

**lemma** *sep-add-zero-sym* [*simp*]: $0 + x = x$
  **by** (*simp add*: *sep-add-commute*)

**lemma** *sep-disj-addD2*: $[\![\ x \mathbin{\#\#} y + z;\ y \mathbin{\#\#} z\ ]\!] \Longrightarrow x \mathbin{\#\#} z$
**by** (*metis sep-disj-addD1 sep-add-ac*)

**lemma** *sep-disj-addD*: $[\![\ x \mathbin{\#\#} y + z;\ y \mathbin{\#\#} z\ ]\!] \Longrightarrow x \mathbin{\#\#} y \wedge x \mathbin{\#\#} z$
**by** (*metis sep-disj-addD1 sep-disj-addD2*)

**lemma** *sep-add-disjD*: $[\![\ x + y \mathbin{\#\#} z;\ x \mathbin{\#\#} y\ ]\!] \Longrightarrow x \mathbin{\#\#} z \wedge y \mathbin{\#\#} z$
**by** (*metis sep-disj-addD sep-disj-commuteI*)

**lemma** *sep-disj-addI2*:
$[\![\ x \mathbin{\#\#} y + z;\ y \mathbin{\#\#} z\ ]\!] \Longrightarrow x + z \mathbin{\#\#} y$
**by** (*metis sep-add-ac sep-disj-addI1*)

**lemma** *sep-add-disjI1*:
$[\![\ x + y \mathbin{\#\#} z;\ x \mathbin{\#\#} y\ ]\!] \Longrightarrow x + z \mathbin{\#\#} y$
**by** (*metis sep-add-ac sep-add-disjD sep-disj-addI2*)

**lemma** *sep-add-disjI2*:
$[\![\ x + y \mathbin{\#\#} z;\ x \mathbin{\#\#} y\ ]\!] \Longrightarrow z + y \mathbin{\#\#} x$
**by** (*metis sep-add-ac sep-add-disjD sep-disj-addI2*)

**lemma** *sep-disj-addI3*:
$x + y \mathbin{\#\#} z \Longrightarrow x \mathbin{\#\#} y \Longrightarrow x \mathbin{\#\#} y + z$
**by** (*metis sep-add-ac sep-add-disjD sep-add-disjI2*)

**lemma** *sep-disj-add*:
$[\![\ y \mathbin{\#\#} z;\ x \mathbin{\#\#} y\ ]\!] \Longrightarrow x \mathbin{\#\#} y + z = x + y \mathbin{\#\#} z$
**by** (*metis sep-disj-addI1 sep-disj-addI3*)

## 4.3 Substate Properties

**lemma** *sep-substate-disj-add*:
$x \mathbin{\#\#} y \Longrightarrow x \preceq x + y$
**unfolding** *sep-substate-def* **by** *blast*

**lemma** *sep-substate-disj-add'*:
$x \mathbin{\#\#} y \Longrightarrow x \preceq y + x$
**by** (*simp add*: *sep-add-ac sep-substate-disj-add*)

## 4.4 Separating Conjunction Properties

**lemma** *sep-conjD*:
$(P \wedge\!* Q)\ h \Longrightarrow \exists x\ y.\ x \mathbin{\#\#} y \wedge h = x + y \wedge P\ x \wedge Q\ y$
**by** (*simp add*: *sep-conj-def*)

**lemma** *sep-conjE*:
$[\![\ (P \mathbin{**} Q)\ h;\ \bigwedge x\ y.\ [\![\ P\ x;\ Q\ y;\ x \mathbin{\#\#} y;\ h = x + y\ ]\!] \Longrightarrow X\ ]\!] \Longrightarrow X$
**by** (*auto simp*: *sep-conj-def*)

**lemma** *sep-conjI*:
  ⟦ *P x*; *Q y*; *x ## y*; *h = x + y* ⟧ ⟹ (*P ** Q*) *h*
  **by** (*auto simp*: *sep-conj-def*)

**lemma** *sep-conj-commuteI*:
  (*P ** Q*) *h* ⟹ (*Q ** P*) *h*
  **by** (*auto intro*!: *sep-conjI elim*!: *sep-conjE simp*: *sep-add-ac*)

**lemma** *sep-conj-commute*:
  (*P ** Q*) = (*Q ** P*)
  **by** (*rule ext*) (*auto intro*: *sep-conj-commuteI*)

**lemma** *sep-conj-assoc*:
  ((*P ** Q*) ** *R*) = (*P ** Q ** R*) (**is** *?lhs = ?rhs*)
**proof** (*rule ext*, *rule iffI*)
  **fix** *h*
  **assume** *a*: *?lhs h*
  **then obtain** *x y z* **where** *P x* **and** *Q y* **and** *R z*
                  **and** *x ## y* **and** *x ## z* **and** *y ## z* **and** *x + y ## z*
                  **and** *h = x + y + z*
    **by** (*auto dest*!: *sep-conjD dest*: *sep-add-disjD*)
  **moreover**
  **then have** *x ## y + z*
    **by** (*simp add*: *sep-disj-add*)
  **ultimately**
  **show** *?rhs h*
    **by** (*auto simp*: *sep-add-ac intro*!: *sep-conjI*)
**next**
  **fix** *h*
  **assume** *a*: *?rhs h*
  **then obtain** *x y z* **where** *P x* **and** *Q y* **and** *R z*
                  **and** *x ## y* **and** *x ## z* **and** *y ## z* **and** *x ## y + z*
                  **and** *h = x + y + z*
    **by** (*fastforce elim*!: *sep-conjE simp*: *sep-add-ac dest*: *sep-disj-addD*)
  **thus** *?lhs h*
    **by** (*metis sep-conj-def sep-disj-addI1*)
**qed**

**lemma** *sep-conj-impl*:
  ⟦ (*P ** Q*) *h*; ⋀*h*. *P h* ⟹ *P' h*; ⋀*h*. *Q h* ⟹ *Q' h* ⟧ ⟹ (*P' ** Q'*) *h*
  **by** (*erule sep-conjE*, *auto intro*!: *sep-conjI*)

**lemma** *sep-conj-impl1*:
  **assumes** *P*: ⋀*h*. *P h* ⟹ *I h*
  **shows** (*P ** R*) *h* ⟹ (*I ** R*) *h*
  **by** (*auto intro*: *sep-conj-impl P*)

**lemma** *sep-globalise*:
  ⟦ (*P ** R*) *h*; (⋀*h*. *P h* ⟹ *Q h*) ⟧ ⟹ (*Q ** R*) *h*

**by** (*fast elim*: *sep-conj-impl*)

**lemma** *sep-conj-trivial-strip2*:
$Q = R \Longrightarrow (Q \mathbin{**} P) = (R \mathbin{**} P)$ **by** *simp*

**lemma** *disjoint-subheaps-exist*:
$\exists x\ y.\ x \mathbin{\#\#} y \wedge h = x + y$
**by** (*rule-tac x=0* **in** *exI*, *auto*)

**lemma** *sep-conj-left-commute*:
$(P \mathbin{**} (Q \mathbin{**} R)) = (Q \mathbin{**} (P \mathbin{**} R))$ (**is** *?x = ?y*)
**proof** −
**have** *?x* = $((Q \mathbin{**} R) \mathbin{**} P)$ **by** (*simp add*: *sep-conj-commute*)
**also have** ... = $(Q \mathbin{**} (R \mathbin{**} P))$ **by** (*subst sep-conj-assoc*, *simp*)
**finally show** *?thesis* **by** (*simp add*: *sep-conj-commute*)
**qed**

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

**lemma** *ab-semigroup-mult-sep-conj*: *class.ab-semigroup-mult* (∗∗)
**by** (*unfold-locales*)
(*auto simp*: *sep-conj-ac*)

**lemma** *sep-empty-zero* [*simp*,*intro*!]: □ *0*
**by** (*simp add*: *sep-empty-def*)

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

**lemma** *sep-conj-sep-true*:
$P\ h \Longrightarrow (P \mathbin{**} \textit{sep-true})\ h$
**by** (*simp add*: *sep-conjI*[**where** *y=0*])

**lemma** *sep-conj-sep-true′*:
$P\ h \Longrightarrow (\textit{sep-true} \mathbin{**} P)\ h$
**by** (*simp add*: *sep-conjI*[**where** *x=0*])

**lemma** *sep-conj-true* [*simp*]:
$(\textit{sep-true} \mathbin{**} \textit{sep-true}) = \textit{sep-true}$
**unfolding** *sep-conj-def*
**by** (*auto intro*!: *ext intro*: *disjoint-subheaps-exist*)

**lemma** *sep-conj-false-right* [*simp*]:
$(P \mathbin{**} \textit{sep-false}) = \textit{sep-false}$
**by** (*force elim*: *sep-conjE intro*!: *ext*)

**lemma** *sep-conj-false-left* [*simp*]:
$(\textit{sep-false} \mathbin{**} P) = \textit{sep-false}$
**by** (*subst sep-conj-commute*) (*rule sep-conj-false-right*)

## 4.6 Properties of zero ($\square$)

**lemma** *sep-conj-empty* [*simp*]:
$(P \mathbin{**} \square) = P$
**by** (*simp add*: *sep-conj-def sep-empty-def*)

**lemma** *sep-conj-empty'*[*simp*]:
$(\square \mathbin{**} P) = P$
**by** (*subst sep-conj-commute*, *rule sep-conj-empty*)

**lemma** *sep-conj-sep-emptyI*:
$P\ h \Longrightarrow (P \mathbin{**} \square)\ h$
**by** *simp*

**lemma** *sep-conj-sep-emptyE*:
$[\![\ P\ s;\ (P \mathbin{**} \square)\ s \Longrightarrow (Q \mathbin{**} R)\ s\ ]\!] \Longrightarrow (Q \mathbin{**} R)\ s$
**by** *simp*

**lemma** *monoid-add*: *class.monoid-add* $((**))\ \square$
**by** (*unfold-locales*) (*auto simp*: *sep-conj-ac*)

**lemma** *comm-monoid-add*: *class.comm-monoid-add* $(**)\ \square$
**by** (*unfold-locales*) (*auto simp*: *sep-conj-ac*)

## 4.7 Properties of top (*sep-true*)

**lemma** *sep-conj-true-P* [*simp*]:
$(\mathit{sep\text{-}true} \mathbin{**} (\mathit{sep\text{-}true} \mathbin{**} P)) = (\mathit{sep\text{-}true} \mathbin{**} P)$
**by** (*simp add*: *sep-conj-assoc*[*symmetric*])

**lemma** *sep-conj-disj*:
$((P\ or\ Q) \mathbin{**} R) = ((P \mathbin{**} R)\ or\ (Q \mathbin{**} R))$
**by** (*auto simp*: *sep-conj-def intro*!: *ext*)

**lemma** *sep-conj-sep-true-left*:
$(P \mathbin{**} Q)\ h \Longrightarrow (\mathit{sep\text{-}true} \mathbin{**} Q)\ h$
**by** (*erule sep-conj-impl*, *simp*+)

**lemma** *sep-conj-sep-true-right*:
$(P \mathbin{**} Q)\ h \Longrightarrow (P \mathbin{**} \mathit{sep\text{-}true})\ h$
**by** (*subst* (*asm*) *sep-conj-commute*, *drule sep-conj-sep-true-left*,
*simp add*: *sep-conj-ac*)

## 4.8 Separating Conjunction with Quantifiers

**lemma** *sep-conj-conj*:
$((P\ and\ Q) \mathbin{**} R)\ h \Longrightarrow ((P \mathbin{**} R)\ and\ (Q \mathbin{**} R))\ h$
**by** (*force intro*: *sep-conjI elim*!: *sep-conjE*)

**lemma** *sep-conj-exists1*:

$((EXS\ x.\ P\ x)\ **\ Q) = (EXS\ x.\ (P\ x\ **\ Q))$
**by** (*force intro*!: *ext intro*: *sep-conjI elim*: *sep-conjE*)

**lemma** *sep-conj-exists2*:
$(P\ **\ (EXS\ x.\ Q\ x)) = (EXS\ x.\ P\ **\ Q\ x)$
**by** (*force intro*!: *sep-conjI ext elim*!: *sep-conjE*)

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

**lemma** *sep-conj-spec*:
$((ALLS\ x.\ P\ x)\ **\ Q)\ h \Longrightarrow (P\ x\ **\ Q)\ h$
**by** (*force intro*: *sep-conjI elim*: *sep-conjE*)

## 4.9 Properties of Separating Implication

**lemma** *sep-implI*:
**assumes** *a*: $\bigwedge h'.\ [\![\ h\ \#\#\ h';\ P\ h'\ ]\!] \Longrightarrow Q\ (h + h')$
**shows** $(P \longrightarrow\!* Q)\ h$
**unfolding** *sep-impl-def* **by** (*auto elim*: *a*)

**lemma** *sep-implD*:
$(x \longrightarrow\!* y)\ h \Longrightarrow \forall h'.\ h\ \#\#\ h' \wedge x\ h' \longrightarrow y\ (h + h')$
**by** (*force simp*: *sep-impl-def*)

**lemma** *sep-implE*:
$(x \longrightarrow\!* y)\ h \Longrightarrow (\forall h'.\ h\ \#\#\ h' \wedge x\ h' \longrightarrow y\ (h + h') \Longrightarrow Q) \Longrightarrow Q$
**by** (*auto dest*: *sep-implD*)

**lemma** *sep-impl-sep-true* [*simp*]:
$(P \longrightarrow\!* sep\text{-}true) = sep\text{-}true$
**by** (*force intro*!: *sep-implI ext*)

**lemma** *sep-impl-sep-false* [*simp*]:
$(sep\text{-}false \longrightarrow\!* P) = sep\text{-}true$
**by** (*force intro*!: *sep-implI ext*)

**lemma** *sep-impl-sep-true-P*:
$(sep\text{-}true \longrightarrow\!* P)\ h \Longrightarrow P\ h$
**by** (*clarsimp dest*!: *sep-implD elim*!: *allE*[**where** *x=0*])

**lemma** *sep-impl-sep-true-false* [*simp*]:
$(sep\text{-}true \longrightarrow\!* sep\text{-}false) = sep\text{-}false$
**by** (*force intro*!: *ext dest*: *sep-impl-sep-true-P*)

**lemma** *sep-conj-sep-impl*:
$[\![\ P\ h;\ \bigwedge h.\ (P\ **\ Q)\ h \Longrightarrow R\ h\ ]\!] \Longrightarrow (Q \longrightarrow\!* R)\ h$
**proof** (*rule sep-implI*)
**fix** $h'\ h$
**assume** $P\ h$ **and** $h\ \#\#\ h'$ **and** $Q\ h'$

**hence** $(P \mathbin{**} Q)\ (h + h')$ **by** (*force intro*: *sep-conjI*)
**moreover assume** $\bigwedge h.\ (P \mathbin{**} Q)\ h \Longrightarrow R\ h$
**ultimately show** $R\ (h + h')$ **by** *simp*
**qed**

**lemma** *sep-conj-sep-impl2*:
$[\![ (P \mathbin{**} Q)\ h;\ \bigwedge h.\ P\ h \Longrightarrow (Q \longrightarrow\!* R)\ h\ ]\!] \Longrightarrow R\ h$
**by** (*force dest*: *sep-implD elim*: *sep-conjE*)

**lemma** *sep-conj-sep-impl-sep-conj2*:
$(P \mathbin{**} R)\ h \Longrightarrow (P \mathbin{**} (Q \longrightarrow\!* (Q \mathbin{**} R)))\ h$
**by** (*erule* (*1*) *sep-conj-impl*, *erule sep-conj-sep-impl*, *simp add*: *sep-conj-ac*)

## 4.10 Pure assertions

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

**lemma** *pure-sep-true*:
*pure sep-true*
**by** (*simp add*: *pure-def*)

**lemma** *pure-sep-false*:
*pure sep-true*
**by** (*simp add*: *pure-def*)

**lemma** *pure-split*:
$pure\ P = (P = sep\text{-}true \vee P = sep\text{-}false)$
**by** (*force simp*: *pure-def intro*!: *ext*)

**lemma** *pure-sep-conj*:
$[\![ pure\ P;\ pure\ Q\ ]\!] \Longrightarrow pure\ (P \wedge\!* Q)$
**by** (*force simp*: *pure-split*)

**lemma** *pure-sep-impl*:
$[\![ pure\ P;\ pure\ Q\ ]\!] \Longrightarrow pure\ (P \longrightarrow\!* Q)$
**by** (*force simp*: *pure-split*)

**lemma** *pure-conj-sep-conj*:
$[\![ (P\ and\ Q)\ h;\ pure\ P \vee pure\ Q\ ]\!] \Longrightarrow (P \wedge\!* Q)\ h$
**by** (*metis pure-def sep-add-zero sep-conjI sep-conj-commute sep-disj-zero*)

**lemma** *pure-sep-conj-conj*:
$[\![ (P \wedge\!* Q)\ h;\ pure\ P;\ pure\ Q\ ]\!] \Longrightarrow (P\ and\ Q)\ h$
**by** (*force simp*: *pure-split*)

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

**by** (*auto simp*: *pure-split*)

**lemma** *pure-sep-impl-impl*:
⟦ (*P* ⟶∗ *Q*) *h*; *pure P* ⟧ ⟹ *P h* ⟶ *Q h*
**by** (*force simp*: *pure-split dest*: *sep-impl-sep-true-P*)

**lemma** *pure-impl-sep-impl*:
⟦ *P h* ⟶ *Q h*; *pure P*; *pure Q* ⟧ ⟹ (*P* ⟶∗ *Q*) *h*
**by** (*force simp*: *pure-split*)

**lemma** *pure-conj-right*: (*Q* ∧∗ (⟨*P'*⟩ *and Q'*)) = (⟨*P'*⟩ *and* (*Q* ∧∗ *Q'*))
**by** (*rule ext*, *rule*, *rule*, *clarsimp elim*!: *sep-conjE*)
(*erule sep-conj-impl*, *auto*)

**lemma** *pure-conj-right'*: (*Q* ∧∗ (*P' and* ⟨*Q'*⟩)) = (⟨*Q'*⟩ *and* (*Q* ∧∗ *P'*))
**by** (*simp add*: *conj-comms pure-conj-right*)

**lemma** *pure-conj-left*: ((⟨*P'*⟩ *and Q'*) ∧∗ *Q*) = (⟨*P'*⟩ *and* (*Q'* ∧∗ *Q*))
**by** (*simp add*: *pure-conj-right sep-conj-ac*)

**lemma** *pure-conj-left'*: ((*P' and* ⟨*Q'*⟩) ∧∗ *Q*) = (⟨*Q'*⟩ *and* (*P'* ∧∗ *Q*))
**by** (*subst conj-comms*, *subst pure-conj-left*, *simp*)

**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* ⇒ *bool*) ⇒ *bool* **where**
*intuitionistic P* ≡ ∀ *h h'*. *P h* ∧ *h* ≼ *h'* ⟶ *P h'*

**lemma** *intuitionisticI*:
(⋀*h h'*. ⟦ *P h*; *h* ≼ *h'* ⟧ ⟹ *P h'*) ⟹ *intuitionistic P*
**by** (*unfold intuitionistic-def*, *fast*)

**lemma** *intuitionisticD*:
⟦ *intuitionistic P*; *P h*; *h* ≼ *h'* ⟧ ⟹ *P h'*
**by** (*unfold intuitionistic-def*, *fast*)

**lemma** *pure-intuitionistic*:
*pure P* ⟹ *intuitionistic P*
**by** (*clarsimp simp*: *intuitionistic-def pure-def*, *fast*)

**lemma** *intuitionistic-conj*:
⟦ *intuitionistic P*; *intuitionistic Q* ⟧ ⟹ *intuitionistic* (*P and Q*)
**by** (*force intro*: *intuitionisticI dest*: *intuitionisticD*)

**lemma** *intuitionistic-disj*:
  ⟦ *intuitionistic P*; *intuitionistic Q* ⟧ ⟹ *intuitionistic* (*P or Q*)
  **by** (*force intro*: *intuitionisticI dest*: *intuitionisticD*)

**lemma** *intuitionistic-forall*:
  (⋀*x*. *intuitionistic* (*P x*)) ⟹ *intuitionistic* (*ALLS x*. *P x*)
  **by** (*force intro*: *intuitionisticI dest*: *intuitionisticD*)

**lemma** *intuitionistic-exists*:
  (⋀*x*. *intuitionistic* (*P x*)) ⟹ *intuitionistic* (*EXS x*. *P x*)
  **by** (*force intro*: *intuitionisticI dest*: *intuitionisticD*)

**lemma** *intuitionistic-sep-conj-sep-true*:
  *intuitionistic* (*sep-true* ∧∗ *P*)
**proof** (*rule intuitionisticI*)
  **fix** *h h′ r*
  **assume** *a*: (*sep-true* ∧∗ *P*) *h*
  **then obtain** *x y* **where** *P*: *P y* **and** *h*: *h* = *x* + *y* **and** *xyd*: *x* ## *y*
    **by** − (*drule sep-conjD*, *clarsimp*)
  **moreover assume** *a2*: *h* ≼ *h′*
  **then obtain** *z* **where** *h′*: *h′* = *h* + *z* **and** *hzd*: *h* ## *z*
    **by** (*clarsimp simp*: *sep-substate-def*)

  **moreover have** (*P* ∧∗ *sep-true*) (*y* + (*x* + *z*))
    **using** *P h hzd xyd*
    **by** (*metis sep-add-disjI1 sep-disj-commute sep-conjI*)
  **ultimately show** (*sep-true* ∧∗ *P*) *h′* **using** *hzd*
    **by** (*auto simp*: *sep-conj-commute sep-add-ac dest*!: *sep-disj-addD*)
**qed**

**lemma** *intuitionistic-sep-impl-sep-true*:
  *intuitionistic* (*sep-true* ⟶∗ *P*)
**proof** (*rule intuitionisticI*)
  **fix** *h h′*
  **assume** *imp*: (*sep-true* ⟶∗ *P*) *h* **and** *hh′*: *h* ≼ *h′*

  **from** *hh′* **obtain** *z* **where** *h′*: *h′* = *h* + *z* **and** *hzd*: *h* ## *z*
    **by** (*clarsimp simp*: *sep-substate-def*)
  **show** (*sep-true* ⟶∗ *P*) *h′* **using** *imp h′ hzd*
    **apply** (*clarsimp dest*!: *sep-implD*)
    **apply** (*metis sep-add-assoc sep-add-disjD sep-disj-addI3 sep-implI*)
    **done**
**qed**

**lemma** *intuitionistic-sep-conj*:
  **assumes** *ip*: *intuitionistic* (*P*::(′*a* ⇒ *bool*))
  **shows** *intuitionistic* (*P* ∧∗ *Q*)
**proof** (*rule intuitionisticI*)
  **fix** *h h′*

**assume** *sc*: $(P \wedge* Q)\ h$ **and** *hh'*: $h \preceq h'$

**from** *hh'* **obtain** *z* **where** *h'*: $h' = h + z$ **and** *hzd*: $h \#\# z$
**by** (*clarsimp simp*: *sep-substate-def*)

**from** *sc* **obtain** *x y* **where** *px*: $P\ x$ **and** *qy*: $Q\ y$
**and** *h*: $h = x + y$ **and** *xyd*: $x \#\# y$
**by** (*clarsimp simp*: *sep-conj-def*)

**have** $x \#\# z$ **using** *hzd h xyd*
**by** (*metis sep-add-disjD*)

**with** *ip px* **have** $P\ (x + z)$
**by** (*fastforce elim*: *intuitionisticD sep-substate-disj-add*)

**thus** $(P \wedge* Q)\ h'$ **using** *h' h hzd qy xyd*
**by** (*metis* (*full-types*) *sep-add-commute sep-add-disjD sep-add-disjI2*
*sep-add-left-commute sep-conjI*)
**qed**

**lemma** *intuitionistic-sep-impl*:
**assumes** *iq*: *intuitionistic Q*
**shows** *intuitionistic* $(P \longrightarrow* Q)$
**proof** (*rule intuitionisticI*)
**fix** *h h'*
**assume** *imp*: $(P \longrightarrow* Q)\ h$ **and** *hh'*: $h \preceq h'$

**from** *hh'* **obtain** *z* **where** *h'*: $h' = h + z$ **and** *hzd*: $h \#\# z$
**by** (*clarsimp simp*: *sep-substate-def*)

**{**
**fix** *x*
**assume** *px*: $P\ x$ **and** *hzx*: $h + z \#\# x$

**have** $h + x \preceq h + x + z$ **using** *hzx hzd*
**by** (*metis sep-add-disjI1 sep-substate-def*)

**with** *imp hzd iq px hzx*
**have** $Q\ (h + z + x)$
**by** (*metis intuitionisticD sep-add-assoc sep-add-ac sep-add-disjD sep-implE*)
**}**

**with** *imp h' hzd iq* **show** $(P \longrightarrow* Q)\ h'$
**by** (*fastforce intro*: *sep-implI*)
**qed**

**lemma** *strongest-intuitionistic*:
$\neg\ (\exists\, Q.\ (\forall\, h.\ (Q\ h \longrightarrow (P \wedge* \textit{sep-true})\ h)) \wedge \textit{intuitionistic}\ Q \wedge$
$Q \neq (P \wedge* \textit{sep-true}) \wedge (\forall\, h.\ P\ h \longrightarrow Q\ h))$

**by** (*fastforce intro*!: *ext sep-substate-disj-add*
*dest*!: *sep-conjD intuitionisticD*)

**lemma** *weakest-intuitionistic*:
¬ (∃ *Q*. (∀ *h*. ((*sep-true* ⟶∗ *P*) *h* ⟶ *Q h*)) ∧ *intuitionistic Q* ∧
*Q* ≠ (*sep-true* ⟶∗ *P*) ∧ (∀ *h*. *Q h* ⟶ *P h*))
**apply** (*clarsimp intro*!: *ext*)
**apply** (*rule iffI*)
**apply** (*rule sep-implI*)
**apply** (*drule-tac h*=*x* **and** *h'*=*x* + *h'* **in** *intuitionisticD*)
**apply** (*clarsimp simp*: *sep-add-ac sep-substate-disj-add*)+
**done**

**lemma** *intuitionistic-sep-conj-sep-true-P*:
⟦ (*P* ∧∗ *sep-true*) *s*; *intuitionistic P* ⟧ ⟹ *P s*
**by** (*force dest*: *intuitionisticD elim*: *sep-conjE sep-substate-disj-add*)

**lemma** *intuitionistic-sep-conj-sep-true-simp*:
*intuitionistic P* ⟹ (*P* ∧∗ *sep-true*) = *P*
**by** (*fast intro*!: *sep-conj-sep-true ext*
*elim*: *intuitionistic-sep-conj-sep-true-P*)

**lemma** *intuitionistic-sep-impl-sep-true-P*:
⟦ *P h*; *intuitionistic P* ⟧ ⟹ (*sep-true* ⟶∗ *P*) *h*
**by** (*force intro*!: *sep-implI dest*: *intuitionisticD*
*intro*: *sep-substate-disj-add*)

**lemma** *intuitionistic-sep-impl-sep-true-simp*:
*intuitionistic P* ⟹ (*sep-true* ⟶∗ *P*) = *P*
**by** (*fast intro*!: *ext*
*elim*: *sep-impl-sep-true-P intuitionistic-sep-impl-sep-true-P*)

## 4.12 Strictly exact assertions

**definition** *strictly-exact* :: ('*a* ⇒ *bool*) ⇒ *bool* **where**
*strictly-exact P* ≡ ∀ *h h'*. *P h* ∧ *P h'* ⟶ *h* = *h'*

**lemma** *strictly-exactD*:
⟦ *strictly-exact P*; *P h*; *P h'* ⟧ ⟹ *h* = *h'*
**by** (*unfold strictly-exact-def*, *fast*)

**lemma** *strictly-exactI*:
(⋀*h h'*. ⟦ *P h*; *P h'* ⟧ ⟹ *h* = *h'*) ⟹ *strictly-exact P*
**by** (*unfold strictly-exact-def*, *fast*)

**lemma** *strictly-exact-sep-conj*:
⟦ *strictly-exact P*; *strictly-exact Q* ⟧ ⟹ *strictly-exact* (*P* ∧∗ *Q*)
**apply** (*rule strictly-exactI*)
**apply** (*erule sep-conjE*)+

**apply** (*drule-tac h=x* **and** *h'=xa* **in** *strictly-exactD*, *assumption+*)
**apply** (*drule-tac h=y* **and** *h'=ya* **in** *strictly-exactD*, *assumption+*)
**apply** *clarsimp*
**done**

**lemma** *strictly-exact-conj-impl*:
⟦ (*Q* ∧∗ *sep-true*) *h*; *P h*; *strictly-exact Q* ⟧ ⟹ (*Q* ∧∗ (*Q* ⟶∗ *P*)) *h*
**by** (*force intro*: *sep-conjI sep-implI dest*: *strictly-exactD elim*!: *sep-conjE*
*simp*: *sep-add-commute sep-add-assoc*)

**end**

**interpretation** *sep*: *ab-semigroup-mult* (∗∗)
**by** (*rule ab-semigroup-mult-sep-conj*)

**interpretation** *sep*: *comm-monoid-add* (∗∗) □
**by** (*rule comm-monoid-add*)

# 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* ⟹ *x* ## *y* + *z* = (*x* ## *y* ∧ *x* ## *z*)
**begin**

**lemma** *sep-disj-add-eq* [*simp*]: *x* ## *y* ⟹ *x* + *y* ## *z* = (*x* ## *z* ∧ *y* ## *z*)
**by** (*metis sep-add-disj-eq sep-disj-commute*)

**subclass** *sep-algebra* **by** *standard auto*

**end**

# 6 Folding separating conjunction over lists of predicates

**lemma** *sep-list-conj-Nil* [*simp*]: ⋀∗ [] = □
**by** (*simp add*: *sep-list-conj-def*)

**lemma** (**in** *semigroup-add*) *foldl-assoc*:
**shows** *foldl* (+) (*x*+*y*) *zs* = *x* + (*foldl* (+) *y zs*)
**by** (*induct zs arbitrary*: *y*) (*simp-all add*:*add.assoc*)

**lemma** (**in** *monoid-add*) *foldl-absorb0*:
**shows** *x* + (*foldl* (+) *0 zs*) = *foldl* (+) *x zs*
**by** (*induct zs*) (*simp-all add*:*foldl-assoc*)

**lemma** *sep-list-conj-Cons* [*simp*]: $\bigwedge*\ (x\#xs) = (x\ **\ \bigwedge*\ xs)$
**by** (*simp add*: *sep-list-conj-def sep.foldl-absorb0*)

**lemma** *sep-list-conj-append* [*simp*]: $\bigwedge*\ (xs\ @\ ys) = (\bigwedge*\ xs\ **\ \bigwedge*\ ys)$
**by** (*simp add*: *sep-list-conj-def sep.foldl-absorb0*)

**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** (*induct xs*)
**case** *Nil* **thus** *?case* **by** *clarsimp*
**next**
**case** (*Cons x xs*)
**hence** *IH*: *foldl* (+) *0* (*map f xs*) =
*foldl* (+) *0* (*map f* (*filter P xs*)) +
*foldl* (+) *0* (*map f* [$x\leftarrow xs$ . $\neg$ *P x*])
**by** (*simp only*: *eq-commute*)

**have** *foldl-Cons'*:
$\bigwedge x\ xs$. *foldl* (+) *0* (*x # xs*) = *x* + (*foldl* (+) *0 xs*)
**by** (*simp*, *subst foldl-absorb0*[*symmetric*], *rule refl*)

**{ assume** *P x*
**hence** *?case* **by** (*auto simp del*: *foldl-Cons simp add*: *foldl-Cons' IH ac-simps*)
**} moreover {**
**assume** $\neg$ *P x*
**hence** *?case* **by** (*auto simp del*: *foldl-Cons simp add*: *foldl-Cons' IH ac-simps*)
**}**
**ultimately show** *?case* **by** *blast*
**qed**

# 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*: $[\![\ x + z = y + z\ ;\ x\ \#\#\ z\ ;\ y\ \#\#\ z\ ]\!] \Longrightarrow x = y$
**begin**

**definition**

*precise* :: ('*a* $\Rightarrow$ *bool*) $\Rightarrow$ *bool* **where**

```
  precise P = (∀ h hp hp'. hp ≼ h ∧ P hp ∧ hp' ≼ h ∧ P hp' ⟶ hp = hp')

lemma precise ((=) s)
  by (metis (full-types) precise-def)

lemma sep-add-cancel:
  x ## z ⟹ y ## z ⟹ (x + z = y + z) = (x = y)
  by (metis sep-add-cancelD)

lemma precise-distribute:
  precise P = (∀ Q R. ((Q and R) ∧* P) = ((Q ∧* P) and (R ∧* P)))
proof (rule iffI)
  assume pp: precise P
  {
    fix Q R
    fix h hp hp' s

    { assume a: ((Q and R) ∧* P) s
      hence ((Q ∧* P) and (R ∧* P)) s
        by (fastforce dest!: sep-conjD elim: sep-conjI)
    }
    moreover
    { assume qs: (Q ∧* P) s and qr: (R ∧* P) s

      from qs obtain x y where sxy: s = x + y and xy: x ## y
                           and x: Q x and y: P y
        by (fastforce dest!: sep-conjD)
      from qr obtain x' y' where sxy': s = x' + y' and xy': x' ## y'
                           and x': R x' and y': P y'
        by (fastforce dest!: sep-conjD)

      from sxy have ys: y ≼ x + y using xy
        by (fastforce simp: sep-substate-disj-add' sep-disj-commute)
      from sxy' have ys': y' ≼ x' + y' using xy'
        by (fastforce simp: sep-substate-disj-add' sep-disj-commute)

      from pp have yy: y = y' using sxy sxy' xy xy' y y' ys ys'
        by (fastforce simp: precise-def)

      hence x = x' using sxy sxy' xy xy'
        by (fastforce dest!: sep-add-cancelD)

      hence ((Q and R) ∧* P) s using sxy x x' yy y' xy'
        by (fastforce intro: sep-conjI)
    }
    ultimately
    have ((Q and R) ∧* P) s = ((Q ∧* P) and (R ∧* P)) s using pp by blast
  }
  thus ∀ Q R. ((Q and R) ∧* P) = ((Q ∧* P) and (R ∧* P)) by (blast intro!: ext)
```

**next**
**assume** *a*: $\forall\, Q\ R.\ ((Q\ and\ R) \wedge\!*\ P) = ((Q \wedge\!*\ P)\ and\ (R \wedge\!*\ P))$
**thus** *precise P*
**proof** (*clarsimp simp*: *precise-def*)
**fix** *h hp hp' Q R*
**assume** *hp*: $hp \preceq h$ **and** *hp'*: $hp' \preceq h$ **and** *php*: *P hp* **and** *php'*: *P hp'*

**obtain** *z* **where** *hhp*: $h = hp + z$ **and** *hpz*: $hp \#\# z$ **using** *hp*
**by** (*clarsimp simp*: *sep-substate-def*)
**obtain** *z'* **where** *hhp'*: $h = hp' + z'$ **and** *hpz'*: $hp' \#\# z'$ **using** *hp'*
**by** (*clarsimp simp*: *sep-substate-def*)

**have** *h-eq*: $z' + hp' = z + hp$ **using** *hhp hhp' hpz hpz'*
**by** (*fastforce simp*: *sep-add-ac*)

**from** *hhp hhp' a hpz hpz' h-eq*
**have** $\forall\, Q\ R.\ ((Q\ and\ R) \wedge\!*\ P)\ (z + hp) = ((Q \wedge\!*\ P)\ and\ (R \wedge\!*\ P))\ (z' + hp')$
**by** (*fastforce simp*: *h-eq sep-add-ac sep-conj-commute*)

**hence** $(((=)\ z\ and\ (=)\ z') \wedge\!*\ P)\ (z + hp) =$
$(((=)\ z \wedge\!*\ P)\ and\ ((=)\ z' \wedge\!*\ P))\ (z' + hp')$ **by** *blast*

**thus** $hp = hp'$ **using** *php php' hpz hpz' h-eq*
**by** (*fastforce dest*!: *iffD2 cong*: *conj-cong*
*simp*: *sep-add-ac sep-add-cancel sep-conj-def*)
**qed**
**qed**

**lemma** *strictly-precise*: *strictly-exact P* $\Longrightarrow$ *precise P*
**by** (*metis precise-def strictly-exactD*)

**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 ≡ {x. f x ≠ none}
end

instantiation option :: (type) opt
begin
  definition none-def [simp]: none ≡ None
  instance ..
end

instantiation fun :: (type, opt) zero
begin
  definition zero-fun-def: 0 ≡ λs. none
  instance ..
end

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

definition
  plus-fun-def: m1 + m2 ≡ λx. if m2 x = none then m1 x else m2 x

definition
  sep-disj-fun-def: sep-disj m1 m2 ≡ domain m1 ∩ domain m2 = {}

instance
  apply standard
       apply (simp add: sep-disj-fun-def domain-def zero-fun-def)
      apply (fastforce simp: sep-disj-fun-def)
     apply (simp add: plus-fun-def zero-fun-def)
    apply (simp add: plus-fun-def sep-disj-fun-def domain-def)
    apply (rule ext)
     apply fastforce
   apply (rule ext)
   apply (simp add: plus-fun-def)
  apply (simp add: sep-disj-fun-def domain-def plus-fun-def)
  apply fastforce
 apply (simp add: sep-disj-fun-def domain-def plus-fun-def)
 apply fastforce
 done

end
```

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

```
lemma domain-conv: domain = dom
  by (rule ext) (simp add: domain-def dom-def)

lemma plus-fun-conv: a + b = a ++ b
  by (auto simp: plus-fun-def map-add-def split: option.splits)
```

**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*
**by** (*simp add*: *sep-conj-ac*)

**end**

# 9 Separation Logic Tactics

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

**ML-file** ‹*sep-tactics.ML*›

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

# 10 Selection (move-to-front) tactics

**method-setup** *sep-select* = ‹
*Scan.lift Parse.int >> (fn n => fn ctxt => SIMPLE-METHOD' (sep-select-tac ctxt n))*
› *Select nth separation conjunct in conclusion*

**method-setup** *sep-select-asm* = ‹
*Scan.lift Parse.int >> (fn n => fn ctxt => SIMPLE-METHOD' (sep-select-asm-tac ctxt n))*
› *Select nth separation conjunct in assumptions*

# 11 Substitution

**method-setup** *sep-subst* = ‹
*Scan.lift (Args.mode asm −− Scan.optional (Args.parens (Scan.repeat Parse.nat)) [0]) −−*
*Attrib.thms >> (fn ((asm, occs), thms) => fn ctxt =>*
*SIMPLE-METHOD' ((if asm then sep-subst-asm-tac else sep-subst-tac) ctxt occs thms))*
›
*single−step substitution after solving one separation logic assumption*

# 12 Forward Reasoning

**method-setup** *sep-drule* = ‹
  *Attrib.thms >> (fn thms => fn ctxt => SIMPLE-METHOD' (sep-dtac ctxt thms))*
› *drule after separating conjunction reordering*

**method-setup** *sep-frule* = ‹
  *Attrib.thms >> (fn thms => fn ctxt => SIMPLE-METHOD' (sep-ftac ctxt thms))*
› *frule after separating conjunction reordering*

# 13 Backward Reasoning

**method-setup** *sep-rule* = ‹
  *Attrib.thms >> (fn thms => fn ctxt => SIMPLE-METHOD' (sep-rtac ctxt thms))*
› *applies rule after separating conjunction reordering*

# 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** ‹
```
fun sep-cancel-smart-tac ctxt tac =
  let fun TRY' tac = tac ORELSE' (K all-tac)
  in
    tac
    ORELSE' (sep-cancel-tac ctxt tac
          THEN' TRY' tac
          THEN' TRY' (resolve-tac ctxt @{thms TrueI}))
    ORELSE' (eresolve-tac ctxt @{thms sep-conj-sep-emptyE}
          THEN' sep-cancel-tac ctxt tac
```

```
            THEN' TRY' tac
            THEN' TRY' (resolve-tac ctxt @{thms TrueI}))
    end;

  fun sep-cancel-smart-tac-rules ctxt etacs =
      sep-cancel-smart-tac ctxt (FIRST' ([assume-tac ctxt] @ etacs));

  val sep-cancel-syntax = Method.sections [
    Args.add -- Args.colon >>
      K (Method.modifier (Named-Theorems.add @{named-theorems sep-cancel})
here )];
›

method-setup sep-cancel = ‹
  sep-cancel-syntax >> (fn - => fn ctxt =>
    let
      val etacs = map (eresolve-tac ctxt o single)
        (rev (Named-Theorems.get ctxt @{named-theorems sep-cancel}));
    in
      SIMPLE-METHOD' (sep-cancel-smart-tac-rules ctxt etacs)
    end)
› Separating conjunction conjunct cancellation
```

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

```
method-setup sep-cancel-blast = ‹
  sep-cancel-syntax >> (fn - => fn ctxt =>
    let
      val rules = rev (Named-Theorems.get ctxt @{named-theorems sep-cancel});
      val tac = Blast.depth-tac (ctxt addIs rules) 10;
    in
      SIMPLE-METHOD' (sep-cancel-smart-tac ctxt tac)
    end)
› Separating conjunction conjunct cancellation using blast

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** $x \mapsto y \equiv \lambda h.\ h = [x \mapsto y]$

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

**definition** *maps-to-ex* :: *nat* ⇒ *heap* ⇒ *bool* (‹- ↦ −› [*56*] *56*)
**where** $x \mapsto - \equiv \exists\, y.\ x \mapsto y$

**lemma** *maps-to-maps-to-ex* [*elim*!]:
$(p \mapsto v)\ s \Longrightarrow (p \mapsto -)\ s$
**by** (*auto simp*: *maps-to-ex-def*)

**lemma** *maps-to-write*:
$(p \mapsto - \ast\ast P)\ H \Longrightarrow (p \mapsto v \ast\ast P)\ (H\ (p \mapsto v))$
**apply** (*clarsimp simp*: *sep-conj-def maps-to-def maps-to-ex-def split*: *option.splits*)
**apply** (*rule-tac x*=*y* **in** *exI*)
**apply** (*auto simp*: *sep-disj-fun-def map-convs map-add-def split*: *option.splits*)
**done**

**lemma** *points-to*:
$(p \mapsto v \ast\ast P)\ H \Longrightarrow the\ (H\ p) = v$
**by** (*auto elim*!: *sep-conjE*
*simp*: *sep-disj-fun-def maps-to-def map-convs map-add-def*
*split*: *option.splits*)

**primrec**
*list* :: *nat* ⇒ *nat list* ⇒ *heap* ⇒ *bool*
**where**
$list\ i\ [] = (\langle i{=}0 \rangle\ and\ \square)$
| $list\ i\ (x\#xs) = (\langle i{=}x \wedge i{\neq}0 \rangle\ and\ (EXS\ j.\ i \mapsto j \ast\ast list\ j\ xs))$

**lemma** *list-empty* [*simp*]:
**shows** $list\ 0\ xs = (\lambda s.\ xs = [] \wedge \square\ s)$
**by** (*cases xs*) *auto*

**lemma** *VARS x y z w h*
$\{(x \mapsto y \ast\ast z \mapsto w)\ h\}$
*SKIP*
$\{x \neq z\}$
**apply** *vcg*
**apply**(*auto elim*!: *sep-conjE simp*: *maps-to-def sep-disj-fun-def domain-conv*)

**done**

**lemma** *VARS H x y z w*
 {(*P* ** *Q*) *H*}
 *SKIP*
 {(*Q* ** *P*) *H*}
  **apply** *vcg*
  **apply**(*simp add*: *sep-conj-commute*)
**done**

**lemma** *VARS H*
 {*p*≠*0* ∧ (*p* ↦ − ** *list q qs*) *H*}
 *H* := *H*(*p* ↦ *q*)
 {*list p* (*p*#*qs*) *H*}
  **apply** *vcg*
  **apply** (*auto intro*: *maps-to-write*)
**done**

**lemma** *VARS H p q r*
  {(*list p Ps* ** *list q Qs*) *H*}
  *WHILE p* ≠ *0*
  *INV* {∃*ps qs*. (*list p ps* ** *list q qs*) *H* ∧ *rev ps* @ *qs* = *rev Ps* @ *Qs*}
  *DO r* := *p*; *p* := *the*(*H p*); *H* := *H*(*r* ↦ *q*); *q* := *r OD*
  {*list q* (*rev Ps* @ *Qs*) *H*}
  **supply** [[*simproc del*: *defined-all*]]
  **apply** *vcg*
    **apply** *fastforce*
   **apply** *clarsimp*
   **apply** (*case-tac ps*, *simp*)
   **apply** (*rename-tac p ps*′)
   **apply** (*clarsimp simp*: *sep-conj-exists sep-conj-ac*)
   **apply** (*sep-subst points-to*)
   **apply** (*rule-tac x* = *ps*′ **in** *exI*)
   **apply** (*rule-tac x* = *p* # *qs* **in** *exI*)
   **apply** (*simp add*: *sep-conj-exists sep-conj-ac*)
   **apply** (*rule exI*)
   **apply** (*sep-rule maps-to-write*)
   **apply** ((*sep-cancel add*: *maps-to-maps-to-ex*)+)[*1*]
  **apply** *clarsimp*
  **done**

**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* **by** (*rule heap-sep-algebra*)

**axiomatization**
  *points-to* :: *p* ⇒ *val* ⇒ *heap* ⇒ *bool* **and**
  *val* :: *heap* ⇒ *p* ⇒ *val*
**where**
  *points-to*: (*points-to p v* ** *P*) *h* ⟹ *val h p* = *v*

**lemma**
  ⟦ *Q2* (*val h p*); (*K* ** *T* ** *blub* ** *P* ** *points-to p v* ** *P* ** *J*) *h* ⟧
   ⟹ *Q* (*val h p*) (*val h p*)
  **apply** (*sep-subst* (*2*) *points-to*)
  **apply** (*sep-subst* (*asm*) *points-to*)
  **apply** (*sep-subst points-to*)
  **oops**

**lemma**
  ⟦ *Q2* (*val h p*); (*K* ** *T* ** *blub* ** *P* ** *points-to p v* ** *P* ** *J*) *h* ⟧
   ⟹ *Q* (*val h p*) (*val h p*)
  **apply** (*sep-drule points-to*)
  **apply** *simp*
  **oops**

**lemma**
  ⟦ *Q2* (*val h p*); (*K* ** *T* ** *blub* ** *P* ** *points-to p v* ** *P* ** *J*) *h* ⟧
   ⟹ *Q* (*val h p*) (*val h p*)
  **apply** (*sep-frule points-to*)
  **apply** *simp*
  **oops**

**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*)
  **apply** (*sep-rule a*)
  **oops**

Example of low-level rewrites

**lemma** ⟦ *unrelated s* ; (*P* ** *Q* ** *R*) *s* ⟧ ⟹ (*A* ** *B* ** *Q* ** *P*) *s*
  **apply** (*tactic* ‹*dresolve-tac* @{*context*} [*mk-sep-select-rule* @{*context*} *true* (*3*, *1*)] *1*›)

**apply** (*tactic ‹resolve-tac @{context} [mk-sep-select-rule @{context} false (4, 2)] 1›*)

**apply** (*erule (1) sep-conj-impl*)
**oops**

Conjunct selection

**lemma** (*A ** B ** Q ** P*) *s*
**apply** (*sep-select 1*)
**apply** (*sep-select 3*)
**apply** (*sep-select 4*)
**oops**

**lemma** ⟦ *also unrelated*; (*A ** B ** Q ** P*) *s* ⟧ ⟹ *unrelated*
**apply** (*sep-select-asm 2*)
**oops**

# 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*
**by** (*sep-cancel add*: *forward*)

**lemma**
**assumes** *forward*: ⋀*s*. *generic s* ⟹ *instance s*
**shows** (*A ** generic ** B*) *s* ⟹ (*instance ** sep-true*) *s*
**by** (*sep-cancel add*: *forward*)

**lemma** ⟦ (*A ** B*) *sa* ; (*A ** Y*) *s* ⟧ ⟹ (*A ** X*) *s*
**apply** (*sep-cancel*)
**oops**

**lemma** ⟦ (*A ** B*) *sa* ; (*A ** Y*) *s* ⟧ ⟹ (λ*s*. (*A ** X*) *s*) *s*
**apply** (*sep-cancel*)
**oops**

**schematic-goal** ⟦ (*B ** A ** C*) *s* ⟧ ⟹ (λ*s*. (*A ** ?X*) *s*) *s*
**by** (*sep-cancel*)

**lemma**
**assumes** *forward*: ⋀*s*. *generic s* ⟹ *instance s*
**shows** ⟦ (*A ** B*) *s* ; (*generic ** Y*) *s* ⟧ ⟹ (*X ** instance*) *s*
**apply** (*sep-cancel add*: *forward*)
**oops**

**lemma**

**assumes** *forward*: $\bigwedge s.\ generic\ s \Longrightarrow instance\ s$
**shows** $generic\ s \Longrightarrow instance\ s$
**by** (*sep-cancel add*: *forward*)

**lemma**
**assumes** *forward*: $\bigwedge s.\ generic\ s \Longrightarrow instance\ s$
**assumes** *forward2*: $\bigwedge s.\ instance\ s \Longrightarrow instance2\ s$
**shows** $generic\ s \Longrightarrow (instance2 \mathbin{**} sep\text{-}true)\ s$
**by** (*sep-cancel-blast add*: *forward forward2*)

**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*: $(None \neq x) = (\exists y.\ x = Some\ y)$ **by** (*cases x*) *auto*

**lemma** *None-com*: $(None = x) = (x = None)$ **by** *fast*

**lemma** *Some-com*: $(Some\ y = x) = (x = Some\ y)$ **by** *fast*

# 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** $\langle\cap_m\rangle$ *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** $\langle$`–$\rangle$ *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*: $(m = Map.empty) = (\forall x.\ m\ x = None)$
**by** (*fastforce intro*!: *ext*)

**lemma** *map-le-empty2* [*simp*]:
$(m \subseteq_m Map.empty) = (m = Map.empty)$
**by** (*auto simp*: *map-le-def intro*: *ext*)

**lemma** *dom-iff*:
$(\exists y.\ m\ x = Some\ y) = (x \in dom\ m)$
**by** *auto*

**lemma** *non-dom-eval*:
$x \notin dom\ m \Longrightarrow m\ x = None$
**by** *auto*

**lemma** *non-dom-eval-eq*:
$x \notin dom\ m = (m\ x = None)$
**by** *auto*

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

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

**lemma** *dom-empty-is-empty*:
$(dom\ m = \{\}) = (m = Map.empty)$
**proof** (*rule iffI*)
**assume** *a*: $dom\ m = \{\}$
**{ assume** $m \neq Map.empty$
**hence** $dom\ m \neq \{\}$
**by** − (*subst* (*asm*) *empty-forall-equiv*, *simp add*: *dom-def*)
**hence** *False* **using** *a* **by** *blast*
**}**
**thus** $m = Map.empty$ **by** *blast*
**next**
**assume** *a*: $m = Map.empty$
**thus** $dom\ m = \{\}$ **by** *simp*
**qed**

**lemma** *map-add-dom-eq*:
$dom\ m = dom\ m' \Longrightarrow m ++ m' = m'$
**by** (*rule ext*) (*auto simp*: *map-add-def split*: *option.splits*)

**lemma** *map-add-right-dom-eq*:
$[\![\ m_0 ++ m_1 = m_0{}' ++ m_1{}';\ dom\ m_1 = dom\ m_1{}'\ ]\!] \Longrightarrow m_1 = m_1{}'$

**unfolding** *map-add-def*
**by** (*rule ext*, *rule ccontr*,
*drule-tac x=x* **in** *fun-cong*, *clarsimp split*: *option.splits*,
*drule sym*, *drule sym*, *force+*)

**lemma** *map-le-same-dom-eq*:
⟦ $m_0 \subseteq_m m_1$ ; *dom* $m_0$ = *dom* $m_1$ ⟧ ⟹ $m_0 = m_1$
**by** (*auto intro*!: *ext simp*: *map-le-def elim*!: *ballE*)

## 19.2 Properties of map restriction

**lemma** *restrict-map-cancel*:
($m$ |` $S$ = $m$ |` $T$) = (*dom* $m \cap S$ = *dom* $m \cap T$)
**by** (*fastforce intro*: *ext dest*: *fun-cong*
*simp*: *restrict-map-def None-not-eq*
*split*: *if-split-asm*)

**lemma** *map-add-restricted-self* [*simp*]:
$m$ ++ $m$ |` $S$ = $m$
**by** (*auto intro*: *ext simp*: *restrict-map-def map-add-def split*: *option.splits*)

**lemma** *map-add-restrict-dom-right* [*simp*]:
($m$ ++ $m'$) |` *dom* $m'$ = $m'$
**by** (*rule ext*, *auto simp*: *restrict-map-def map-add-def split*: *option.splits*)

**lemma** *restrict-map-UNIV* [*simp*]:
$m$ |` *UNIV* = $m$
**by** (*simp add*: *restrict-map-def*)

**lemma** *restrict-map-dom*:
$S$ = *dom* $m$ ⟹ $m$ |` $S$ = $m$
**by** (*auto intro*!: *ext simp*: *restrict-map-def None-not-eq*)

**lemma** *restrict-map-subdom*:
*dom* $m \subseteq S$ ⟹ $m$ |` $S$ = $m$
**by** (*fastforce simp*: *restrict-map-def None-com intro*: *ext*)

**lemma** *map-add-restrict*:
($m_0$ ++ $m_1$) |` $S$ = (($m_0$ |` $S$) ++ ($m_1$ |` $S$))
**by** (*force simp*: *map-add-def restrict-map-def intro*: *ext*)

**lemma** *map-le-restrict*:
$m \subseteq_m m'$ ⟹ $m$ = $m'$ |` *dom* $m$
**by** (*force simp*: *map-le-def restrict-map-def None-com intro*: *ext*)

**lemma** *restrict-map-le*:
$m$ |` $S \subseteq_m m$
**by** (*auto simp*: *map-le-def*)

**lemma** *restrict-map-remerge*:
$[\![ S \cap T = \{\} ]\!] \Longrightarrow m \mid` S ++ m \mid` T = m \mid` (S \cup T)$
**by** (*rule ext*, *clarsimp simp*: *restrict-map-def map-add-def*
*split*: *option.splits*)

**lemma** *restrict-map-empty*:
$dom\ m \cap S = \{\} \Longrightarrow m \mid` S = Map.empty$
**by** (*fastforce simp*: *restrict-map-def intro*: *ext*)

**lemma** *map-add-restrict-comp-right* [*simp*]:
$(m \mid` S ++ m \mid` (UNIV - S)) = m$
**by** (*force simp*: *map-add-def restrict-map-def split*: *option.splits intro*: *ext*)

**lemma** *map-add-restrict-comp-right-dom* [*simp*]:
$(m \mid` S ++ m \mid` (dom\ m - S)) = m$
**by** (*auto simp*: *map-add-def restrict-map-def split*: *option.splits intro*!: *ext*)

**lemma** *map-add-restrict-comp-left* [*simp*]:
$(m \mid` (UNIV - S) ++ m \mid` S) = m$
**by** (*subst map-add-comm*, *auto*)

**lemma** *restrict-self-UNIV*:
$m \mid` (dom\ m - S) = m \mid` (UNIV - S)$
**by** (*auto intro*!: *ext simp*: *restrict-map-def*)

**lemma** *map-add-restrict-nonmember-right*:
$x \notin dom\ m' \Longrightarrow (m ++ m') \mid` \{x\} = m \mid` \{x\}$
**by** (*rule ext*, *auto simp*: *restrict-map-def map-add-def split*: *option.splits*)

**lemma** *map-add-restrict-nonmember-left*:
$x \notin dom\ m \Longrightarrow (m ++ m') \mid` \{x\} = m' \mid` \{x\}$
**by** (*rule ext*, *auto simp*: *restrict-map-def map-add-def split*: *option.splits*)

**lemma** *map-add-restrict-right*:
$x \subseteq dom\ m' \Longrightarrow (m ++ m') \mid` x = m' \mid` x$
**by** (*rule ext*, *auto simp*: *restrict-map-def map-add-def split*: *option.splits*)

**lemma** *restrict-map-compose*:
$[\![ S \cup T = dom\ m\ ;\ S \cap T = \{\} ]\!] \Longrightarrow m \mid` S ++ m \mid` T = m$
**by** (*fastforce intro*: *ext simp*: *map-add-def restrict-map-def*)

**lemma** *map-le-dom-subset-restrict*:
$[\![ m' \subseteq_m m;\ dom\ m' \subseteq S ]\!] \Longrightarrow m' \subseteq_m (m \mid` S)$
**by** (*force simp*: *restrict-map-def map-le-def*)

**lemma** *map-le-dom-restrict-sub-add*:
$m' \subseteq_m m \Longrightarrow m \mid` (dom\ m - dom\ m') ++ m' = m$
**by** (*auto simp*: *None-com map-add-def restrict-map-def map-le-def*
*split*: *option.splits*

*intro*!: *ext*)
(*force simp*: *Some-com*)+

**lemma** *subset-map-restrict-sub-add*:
$T \subseteq S \Longrightarrow m \mid` (S - T) ++ m \mid` T = m \mid` S$
**by** (*auto simp*: *restrict-map-def map-add-def intro*!: *ext split*: *option.splits*)

**lemma** *restrict-map-sub-union*:
$m \mid` (dom\ m - (S \cup T)) = (m \mid` (dom\ m - T)) \mid` (dom\ m - S)$
**by** (*auto intro*!: *ext simp*: *restrict-map-def*)

**lemma** *prod-restrict-map-add*:
$[\![ S \cup T = U; S \cap T = \{\} ]\!] \Longrightarrow m \mid` (X \times S) ++ m \mid` (X \times T) = m \mid` (X \times U)$
**by** (*auto simp*: *map-add-def restrict-map-def intro*!: *ext split*: *option.splits*)

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

## 20.1 Definitions

Map disjuction

**definition**
$map\text{-}disj :: ('a \rightharpoonup 'b) \Rightarrow ('a \rightharpoonup 'b) \Rightarrow bool$ (**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*: $h \mid` S \perp h \mathbin{`-} S$
**by** (*fastforce simp*: *sub-restrict-map-def restrict-map-def map-disj-def*
*split*: *option.splits if-split-asm*)

**lemma** *restrict-map-sub-add*: $h \mid` S ++ h \mathbin{`-} S = h$
**by** (*fastforce simp*: *sub-restrict-map-def restrict-map-def map-add-def*
*split*: *option.splits if-split*
*intro*: *ext*)

## 20.3 Properties of map disjunction

**lemma** *map-disj-empty-right* [*simp*]:
$h \perp Map.empty$
**by** (*simp add*: *map-disj-def*)

**lemma** *map-disj-empty-left* [*simp*]:
$Map.empty \perp h$
**by** (*simp add*: *map-disj-def*)

**lemma** *map-disj-com*:
$h_0 \perp h_1 = h_1 \perp h_0$
**by** (*simp add*: *map-disj-def*, *fast*)

**lemma** *map-disjD*:
$h_0 \perp h_1 \Longrightarrow dom\ h_0 \cap dom\ h_1 = \{\}$
**by** (*simp add*: *map-disj-def*)

**lemma** *map-disjI*:
$dom\ h_0 \cap dom\ h_1 = \{\} \Longrightarrow h_0 \perp h_1$
**by** (*simp add*: *map-disj-def*)

## 20.4 Map associativity-commutativity based on map disjuction

**lemma** *map-add-com*:
$h_0 \perp h_1 \Longrightarrow h_0 +\!\!+ h_1 = h_1 +\!\!+ h_0$
**by** (*drule map-disjD*, *rule map-add-comm*, *force*)

**lemma** *map-add-left-commute*:
$h_0 \perp h_1 \Longrightarrow h_0 +\!\!+ (h_1 +\!\!+ h_2) = h_1 +\!\!+ (h_0 +\!\!+ h_2)$
**by** (*simp add*: *map-add-com map-disj-com map-add-assoc*)

**lemma** *map-add-disj*:
$h_0 \perp (h_1 +\!\!+ h_2) = (h_0 \perp h_1 \wedge h_0 \perp h_2)$
**by** (*simp add*: *map-disj-def*, *fast*)

**lemma** *map-add-disj'*:
$(h_1 +\!\!+ h_2) \perp h_0 = (h_1 \perp h_0 \wedge h_2 \perp h_0)$
**by** (*simp add*: *map-disj-def*, *fast*)

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*:
$[\![\ h_0 \perp h_1\ ;\ x \in dom\ h_0\ ]\!] \Longrightarrow h_1\ x = None$
**by** (*auto simp*: *map-disj-def dom-def*)

**lemma** *map-disj-None-left*:
⟦ $h_0 \perp h_1$ ; $x \in dom\ h_1$ ⟧ ⟹ $h_0\ x = None$
**by** (*auto simp*: *map-disj-def dom-def*)

**lemma** *map-disj-None-left'*:
⟦ $h_0\ x = Some\ y$ ; $h_1 \perp h_0$ ⟧ ⟹ $h_1\ x = None$
**by** (*auto simp*: *map-disj-def*)

**lemma** *map-disj-None-right'*:
⟦ $h_1\ x = Some\ y$ ; $h_1 \perp h_0$ ⟧ ⟹ $h_0\ x = None$
**by** (*auto simp*: *map-disj-def*)

**lemma** *map-disj-common*:
⟦ $h_0 \perp h_1$ ; $h_0\ p = Some\ v$ ; $h_1\ p = Some\ v'$ ⟧ ⟹ *False*
**by** (*frule* (*1*) *map-disj-None-left'*, *simp*)

**lemma** *map-disj-eq-dom-left*:
⟦ $h_0 \perp h_1$ ; $dom\ h_0' = dom\ h_0$ ⟧ ⟹ $h_0' \perp h_1$
**by** (*auto simp*: *map-disj-def*)

## 20.6 Map disjunction and addition

**lemma** *map-add-eval-left*:
⟦ $x \in dom\ h$ ; $h \perp h'$ ⟧ ⟹ $(h +\!\!+ h')\ x = h\ x$
**by** (*auto dest*!: *map-disj-None-right simp*: *map-add-def cong*: *option.case-cong*)

**lemma** *map-add-eval-right*:
⟦ $x \in dom\ h'$ ; $h \perp h'$ ⟧ ⟹ $(h +\!\!+ h')\ x = h'\ x$
**by** (*auto elim*!: *map-disjD simp*: *map-add-comm map-add-eval-left map-disj-com*)

**lemma** *map-add-eval-left'*:
⟦ $x \notin dom\ h'$ ; $h \perp h'$ ⟧ ⟹ $(h +\!\!+ h')\ x = h\ x$
**by** (*clarsimp simp*: *map-disj-def map-add-def split*: *option.splits*)

**lemma** *map-add-eval-right'*:
⟦ $x \notin dom\ h$ ; $h \perp h'$ ⟧ ⟹ $(h +\!\!+ h')\ x = h'\ x$
**by** (*clarsimp simp*: *map-disj-def map-add-def split*: *option.splits*)

**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** −
**from** *eq* **have** $h_1 +\!\!+ h_0 = h_1' +\!\!+ h_0'$ **using** *etc* **by** (*simp add*: *map-add-ac*)
**thus** *?thesis* **using** *etc*
**by** (*fastforce elim*!: *map-add-right-dom-eq simp*: *map-add-ac*)
**qed**

**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** (*rule ext*)
**fix** $x$
**from** *eq* **have** *eq'*: $(h_0 ++ h)\ x = (h_1 ++ h)\ x$ **by** (*auto intro*!: *ext*)
**{ assume** $x \in dom\ h$
**hence** $h_0\ x = h_1\ x$ **using** *disj* **by** (*simp add*: *map-disj-None-left*)
**} moreover {**
**assume** $x \notin dom\ h$
**hence** $h_0\ x = h_1\ x$ **using** *disj eq'* **by** (*simp add*: *map-add-eval-left'*)
**}**
**ultimately show** $h_0\ x = h_1\ x$ **by** *cases*
**qed**

**lemma** *map-add-right-eq*:
$[\![h ++ h_0 = h ++ h_1;\ h_0 \perp h;\ h_1 \perp h]\!] \Longrightarrow h_0 = h_1$
**by** (*rule-tac h=h* **in** *map-add-left-eq*, *auto simp*: *map-add-ac*)

**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** (*rule ext*)
**fix** $x$
**from** *merge* **have** *merge-x*: $(h_0 ++ h_1)\ x = (h_0' ++ h_1')\ x$ **by** *simp*
**with** *d ab-disj cd-disj* **show** $h_1\ x = h_1'\ x$
**by** − (*case-tac* $h_1\ x$, *case-tac* $h_1'\ x$, *simp*, *fastforce simp*: *map-disj-def*,
*case-tac* $h_1'\ x$, *clarsimp*, *simp add*: *Some-com*,
*force simp*: *map-disj-def*, *simp*)
**qed**

**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** −
**have** $h_1 ++ h_0 = h_1' ++ h_0'$ **using** *add disj* **by** (*simp add*: *map-add-ac*)
**thus** *?thesis* **using** *dom disj*
**by** − (*rule map-disj-add-eq-dom-right-eq*, *auto simp*: *map-disj-com*)
**qed**

**lemma** *map-add-left-cancel*:
**assumes** *disj*: $h_0 \perp h_1$ $h_0 \perp h_1'$
**shows** $(h_0 ++ h_1 = h_0 ++ h_1') = (h_1 = h_1')$
**proof** (*rule iffI*, *rule ext*)
**fix** $x$

**assume** $(h_0 +\!\!+ h_1) = (h_0 +\!\!+ h_1')$
**hence** $(h_0 +\!\!+ h_1)\ x = (h_0 +\!\!+ h_1')\ x$ **by** (*auto intro*!: *ext*)
**hence** $h_1\ x = h_1'\ x$ **using** *disj*
**by** − (*cases* $x \in dom\ h_0$,
*simp-all add*: *map-disj-None-right map-add-eval-right'*)
**thus** $h_1\ x = h_1'\ x$ **by** (*auto intro*!: *ext*)
**qed** *auto*

**lemma** *map-add-lr-disj*:
$[\![\ h_0 +\!\!+ h_1 = h_0' +\!\!+ h_1';\ h_1 \perp h_1'\ ]\!] \Longrightarrow dom\ h_1 \subseteq dom\ h_0'$
**by** (*clarsimp simp*: *map-disj-def map-add-def*, *drule-tac* $x{=}x$ **in** *fun-cong*)
(*auto split*: *option.splits*)

## 20.7 Map disjunction and map updates

**lemma** *map-disj-update-left* [*simp*]:
$p \in dom\ h_1 \Longrightarrow h_0 \perp h_1(p \mapsto v) = h_0 \perp h_1$
**by** (*clarsimp simp add*: *map-disj-def*, *blast*)

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

**lemma** *map-add-update-left*:
$[\![\ h_0 \perp h_1\ ;\ p \in dom\ h_0\ ]\!] \Longrightarrow (h_0 +\!\!+ h_1)(p \mapsto v) = (h_0(p \mapsto v) +\!\!+ h_1)$
**by** (*drule* (*1*) *map-disj-None-right*)
(*auto intro*: *ext simp*: *map-add-def cong*: *option.case-cong*)

**lemma** *map-add-update-right*:
$[\![\ h_0 \perp h_1\ ;\ p \in dom\ h_1\ ]\!] \Longrightarrow (h_0 +\!\!+ h_1)(p \mapsto v) = (h_0 +\!\!+ h_1\ (p \mapsto v))$
**by** (*drule* (*1*) *map-disj-None-left*)
(*auto intro*: *ext simp*: *map-add-def cong*: *option.case-cong*)

**lemma** *map-add3-update*:
$[\![\ h_0 \perp h_1\ ;\ h_1\ \perp h_2\ ;\ h_0 \perp h_2\ ;\ p \in dom\ h_0\ ]\!]$
$\Longrightarrow (h_0 +\!\!+ h_1 +\!\!+ h_2)(p \mapsto v) = h_0(p \mapsto v) +\!\!+ h_1 +\!\!+ h_2$
**by** (*auto simp*: *map-add-update-left*[*symmetric*] *map-add-ac*)

## 20.8 Map disjunction and $(\subseteq_m)$

**lemma** *map-le-override* [*simp*]:
$[\![\ h \perp h'\ ]\!] \Longrightarrow h \subseteq_m h +\!\!+ h'$
**by** (*auto simp*: *map-le-def map-add-def map-disj-def split*: *option.splits*)

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

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

**lemma** *map-disj-map-le*:
⟦ $h_0{}' \subseteq_m h_0$; $h_0 \perp h_1$ ⟧ ⟹ $h_0{}' \perp h_1$
**by** (*force simp*: *map-disj-def map-le-def*)

**lemma** *map-le-on-disj-left*:
⟦ $h' \subseteq_m h$ ; $h_0 \perp h_1$ ; $h' = h_0 +\!\!+ h_1$ ⟧ ⟹ $h_0 \subseteq_m h$
**unfolding** *map-le-def*
**by** (*rule ballI*, *erule-tac x=a* **in** *ballE*, *auto simp*: *map-add-eval-left*)+

**lemma** *map-le-on-disj-right*:
⟦ $h' \subseteq_m h$ ; $h_0 \perp h_1$ ; $h' = h_1 +\!\!+ h_0$ ⟧ ⟹ $h_0 \subseteq_m h$
**by** (*auto simp*: *map-le-on-disj-left map-add-ac*)

**lemma** *map-le-add-cancel*:
⟦ $h_0 \perp h_1$ ; $h_0{}' \subseteq_m h_0$ ⟧ ⟹ $h_0{}' +\!\!+ h_1 \subseteq_m h_0 +\!\!+ h_1$
**by** (*auto simp*: *map-le-def map-add-def map-disj-def split*: *option.splits*)

**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** $h_0{}' \subseteq_m h_0$
**unfolding** *map-le-def*
**proof** (*rule ballI*)
**fix** *a*
**assume** *a*: $a \in dom\ h_0{}'$
**hence** *sumeq*: $(h_0{}' +\!\!+ h_1)\ a = (h_0 +\!\!+ h_1)\ a$
**using** *subm* **unfolding** *map-le-def* **by** *auto*
**from** *a* **have** $a \notin dom\ h_1$ **using** *disj'* **by** (*auto dest*!: *map-disj-None-right*)
**thus** $h_0{}'\ a = h_0\ a$ **using** *a sumeq disj disj'*
**by** (*simp add*: *map-add-eval-left map-add-eval-left'*)
**qed**

**lemma** *map-le-conv*:
$(h_0{}' \subseteq_m h_0 \wedge h_0{}' \neq h_0) = (\exists h_1.\ h_0 = h_0{}' +\!\!+ h_1 \wedge h_0{}' \perp h_1 \wedge h_0{}' \neq h_0)$
**unfolding** *map-le-def map-disj-def map-add-def*
**by** (*rule iffI*,
*clarsimp intro*!: *exI*[**where** $x=\lambda x.$ *if* $x \notin dom\ h_0{}'$ *then* $h_0\ x$ *else None*])
(*fastforce intro*: *ext intro*: *split*: *option.splits if-split-asm*)+

**lemma** *map-le-conv2*:
$h_0{}' \subseteq_m h_0 = (\exists h_1.\ h_0 = h_0{}' +\!\!+ h_1 \wedge h_0{}' \perp h_1)$
**by** (*case-tac* $h_0{}'=h_0$, *insert map-le-conv*, *auto intro*: *exI*[**where** $x=Map.empty$])

## 20.9 Map disjunction and restriction

**lemma** *map-disj-comp* [*simp*]:
$h_0 \perp h_1 \mid` (UNIV - dom\ h_0)$
**by** (*force simp*: *map-disj-def*)

**lemma** *restrict-map-disj*:
$S \cap T = \{\} \Longrightarrow h \mid` S \perp h \mid` T$
**by** (*auto simp*: *map-disj-def restrict-map-def dom-def*)

**lemma** *map-disj-restrict-dom* [*simp*]:
$h_0 \perp h_1 \mid` (dom\ h_1 - dom\ h_0)$
**by** (*force simp*: *map-disj-def*)

**lemma** *restrict-map-disj-dom-empty*:
$h \perp h' \Longrightarrow h \mid` dom\ h' = Map.empty$
**by** (*fastforce simp*: *map-disj-def restrict-map-def intro*: *ext*)

**lemma** *restrict-map-univ-disj-eq*:
$h \perp h' \Longrightarrow h \mid` (UNIV - dom\ h') = h$
**by** (*rule ext*, *auto simp*: *map-disj-def restrict-map-def*)

**lemma** *restrict-map-disj-dom*:
$h_0 \perp h_1 \Longrightarrow h \mid` dom\ h_0 \perp h \mid` dom\ h_1$
**by** (*auto simp*: *map-disj-def restrict-map-def dom-def*)

**lemma** *map-add-restrict-dom-left*:
$h \perp h' \Longrightarrow (h +\!\!+ h') \mid` dom\ h = h$
**by** (*rule ext*, *auto simp*: *restrict-map-def map-add-def dom-def map-disj-def*
*split*: *option.splits*)

**lemma** *map-add-restrict-dom-left'*:
$h \perp h' \Longrightarrow S = dom\ h \Longrightarrow (h +\!\!+ h') \mid` S = h$
**by** (*rule ext*, *auto simp*: *restrict-map-def map-add-def dom-def map-disj-def*
*split*: *option.splits*)

**lemma** *restrict-map-disj-left*:
$h_0 \perp h_1 \Longrightarrow h_0 \mid` S \perp h_1$
**by** (*auto simp*: *map-disj-def*)

**lemma** *restrict-map-disj-right*:
$h_0 \perp h_1 \Longrightarrow h_0 \perp h_1 \mid` S$
**by** (*auto simp*: *map-disj-def*)

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

**lemma** *map-dom-disj-restrict-right*:
$h_0 \perp h_1 \Longrightarrow (h_0 +\!\!+ h_0') \mid` dom\ h_1 = h_0' \mid` dom\ h_1$
**by** (*simp add*: *map-add-restrict restrict-map-empty map-disj-def*)

**lemma** *restrict-map-on-disj*:
$h_0' \perp h_1 \Longrightarrow h_0 \mid` dom\ h_0' \perp h_1$
**unfolding** *map-disj-def* **by** *auto*

**lemma** *restrict-map-on-disj'*:
$h_0 \perp h_1 \Longrightarrow h_0 \perp h_1 \mid` S$
**by** (*auto simp*: *map-disj-def map-add-def*)

**lemma** *map-le-sub-dom*:
$[\![ h_0 +\!+ h_1 \subseteq_m h \; ; \; h_0 \perp h_1 ]\!] \Longrightarrow h_0 \subseteq_m h \mid` (dom\ h - dom\ h_1)$
**by** (*rule map-le-override-bothD*, *subst map-le-dom-restrict-sub-add*)
(*auto elim*: *map-add-le-mapE simp*: *map-add-ac*)

**lemma** *map-submap-break*:
$[\![ h \subseteq_m h' ]\!] \Longrightarrow h' = (h' \mid` (UNIV - dom\ h)) +\!+ h$
**by** (*fastforce intro*!: *ext split*: *option.splits*
*simp*: *map-le-restrict restrict-map-def map-le-def map-add-def*
*dom-def*)

**lemma** *map-add-disj-restrict-both*:
$[\![ h_0 \perp h_1; S \cap S' = \{\}; T \cap T' = \{\} ]\!]$
$\Longrightarrow (h_0 \mid` S) +\!+ (h_1 \mid` T) \perp (h_0 \mid` S') +\!+ (h_1 \mid` T')$
**by** (*auto simp*: *map-add-ac intro*!: *restrict-map-disj-both restrict-map-disj*)

**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* $((('p \times 'v) \rightharpoonup 'value) \times 'r\ set)$

**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**
**apply** *standard*
**apply** (*simp add*: *zero-vm-sep-state-def sep-disj-vm-sep-state-def*)
**apply** (*fastforce simp*: *sep-disj-vm-sep-state-def map-disj-def*)
**apply** (*case-tac x*, *clarsimp simp*: *zero-vm-sep-state-def*)
**apply** (*case-tac x*, *case-tac y*)
**apply** (*fastforce simp*: *sep-disj-vm-sep-state-def map-add-ac*)
**apply** (*case-tac x*, *case-tac y*, *case-tac z*)
**apply** (*fastforce simp*: *sep-disj-vm-sep-state-def*)
**apply** (*case-tac x*, *case-tac y*, *case-tac z*)
**apply** (*fastforce simp*: *sep-disj-vm-sep-state-def map-add-disj*)
**apply** (*case-tac x*, *case-tac y*, *case-tac z*)
**apply** (*fastforce simp*: *sep-disj-vm-sep-state-def map-add-disj map-disj-com*)
**done**

**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*
  **by** (*auto simp*: *disjoint-def add-comm*)

**lemma** *disj-zero* [*simp*]: *x* ## *0*
  **by** (*auto simp*: *disjoint-def*)

**lemma** *disj-zero2* [*simp*]: *0* ## *x*
  **by** (*subst disj-com*) *simp*

**lemma** *add-zero2* [*simp*]: *0* ⊕ *x* = *Some x*
  **by** (*subst add-comm*) *auto*

**definition**
  *substate* :: *'a* => *'a* => *bool* (**infix** ‹⪯› *60*) **where**
  *a* ⪯ *b* ≡ ∃ *c*. *a* ⊕ *c* = *Some b*

**definition**
  *sep-conj* :: (*'a* ⇒ *bool*) ⇒ (*'a* ⇒ *bool*) ⇒ (*'a* ⇒ *bool*) (**infixl** ‹**›› *61*)
  **where**
  *P* ** *Q* ≡ λ*s*. ∃ *p q*. *p* ⊕ *q* = *Some s* ∧ *P p* ∧ *Q q*

**definition** *emp* :: *'a* ⇒ *bool* (‹□›) **where**
  □ ≡ λ*s*. *s* = *0*

**definition**
*sep-impl* :: (*'a* ⇒ *bool*) ⇒ (*'a* ⇒ *bool*) ⇒ (*'a* ⇒ *bool*) (**infixr** ‹⟶*›  *25*)
**where**
*P* ⟶* *Q* ≡ λ*h*. ∀ *h'* *h''*. *h* ⊕ *h'* = *Some* *h''* ∧ *P* *h'* ⟶ *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*
**by** (*simp add*: *lift2-def add-comm split*: *option.splits*)

**lemma** *add2-None* [*simp*]:
*x* ++ *None* = *None*
**by** (*simp add*: *lift2-def split*: *option.splits*)

**lemma** *None-add2* [*simp*]:
*None* ++ *x* = *None*
**by** (*simp add*: *lift2-def split*: *option.splits*)

**lemma** *add2-Some-Some*:
*Some x* ++ *Some y* = *x* ⊕ *y*
**by** (*simp add*: *lift2-def*)

**lemma** *add2-zero* [*simp*]:
*Some x* ++ *Some 0* = *Some x*
**by** (*simp add*: *add2-Some-Some*)

**lemma** *zero-add2* [*simp*]:
*Some 0* ++ *Some x* = *Some x*
**by** (*simp add*: *add2-Some-Some*)

**lemma** *sep-conjE*:
⟦ (*P* ** *Q*) *s*; ⋀*p* *q*. ⟦ *P* *p*; *Q* *q*; *p* ⊕ *q* = *Some* *s* ⟧ ⟹ *X* ⟧ ⟹ *X*
**by** (*auto simp*: *sep-conj-def*)

**lemma** *sep-conjI*:

⟦ $P\ p$; $Q\ q$; $p \oplus q = Some\ s$ ⟧ ⟹ $(P ** Q)\ s$
**by** (*auto simp*: *sep-conj-def*)

**lemma** *sep-conj-comI*:
$(P ** Q)\ s \Longrightarrow (Q ** P)\ s$
**by** (*auto intro*!: *sep-conjI elim*!: *sep-conjE simp*: *add-comm*)

**lemma** *sep-conj-com*:
$P ** Q = Q ** P$
**by** (*auto intro*: *sep-conj-comI intro*!: *ext*)

**lemma** *lift-to-add2*:
⟦$z \oplus q = Some\ s$; $x \oplus y = Some\ q$⟧ ⟹ $Some\ z +\!\!+ Some\ x +\!\!+ Some\ y = Some\ s$
**by** (*simp add*: *add2-Some-Some*)

**lemma** *lift-to-add2'*:
⟦$q \oplus z = Some\ s$; $x \oplus y = Some\ q$⟧ ⟹ $(Some\ x +\!\!+ Some\ y) +\!\!+ Some\ z = Some\ s$
**by** (*simp add*: *add2-Some-Some*)

**lemma** *add2-Some*:
$(x +\!\!+ Some\ y = Some\ z) = (\exists x'.\ x = Some\ x' \wedge x' \oplus y = Some\ z)$
**by** (*simp add*: *lift2-def split*: *option.splits*)

**lemma** *Some-add2*:
$(Some\ x +\!\!+ y = Some\ z) = (\exists y'.\ y = Some\ y' \wedge x \oplus y' = Some\ z)$
**by** (*simp add*: *lift2-def split*: *option.splits*)

**lemma** *sep-conj-assoc*:
$P ** (Q ** R) = (P ** Q) ** R$
**unfolding** *sep-conj-def*
**apply** (*rule ext*)
**apply** (*rule iffI*)
**apply** *clarsimp*
**apply** (*drule* (*1*) *lift-to-add2*)
**apply** (*subst* (*asm*) *add-assoc*)
**apply** (*fastforce simp*: *add2-Some-Some add2-Some*)
**apply** *clarsimp*
**apply** (*drule* (*1*) *lift-to-add2'*)
**apply** (*subst* (*asm*) *add-assoc* [*symmetric*])
**apply** (*fastforce simp*: *add2-Some-Some Some-add2*)
**done**

**lemma** (**in** −) *sep-true*[*simp*]: *sep-true s* **by** (*simp add*: *sep-true-def*)
**lemma** (**in** −) *sep-false*[*simp*]: ¬*sep-false x* **by** (*simp add*: *sep-false-def*)

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

**by** (*auto simp*: *sep-conjI* [**where** *q=0*])

**lemma** *sep-conj-sep-true′*:
*P s* $\Longrightarrow$ (*sep-true* $**$ *P*) *s*
**by** (*auto simp*: *sep-conjI* [**where** *p=0*])

**lemma** *disjoint-submaps-exist*:
$\exists h_0\ h_1.\ h_0 \oplus h_1 = Some\ h$
**by** (*rule-tac x=0* **in** *exI*, *auto*)

**lemma** *sep-conj-true*[*simp*]:
(*sep-true* $**$ *sep-true*) = *sep-true*
**unfolding** *sep-conj-def*
**by** (*auto intro*!: *ext intro*: *disjoint-submaps-exist*)

**lemma** *sep-conj-false-right*[*simp*]:
(*P* $**$ *sep-false*) = *sep-false*
**by** (*force elim*: *sep-conjE intro*!: *ext*)

**lemma** *sep-conj-false-left*[*simp*]:
(*sep-false* $**$ *P*) = *sep-false*
**by** (*subst sep-conj-com*) (*rule sep-conj-false-right*)

**lemma** *sep-conj-left-com*:
(*P* $**$ (*Q* $**$ *R*)) = (*Q* $**$ (*P* $**$ *R*)) (**is** *?x* = *?y*)
**proof** −
**have** *?x* = ((*Q* $**$ *R*) $**$ *P*) **by** (*simp add*: *sep-conj-com*)
**also have** ... = (*Q* $**$ (*R* $**$ *P*)) **by** (*subst sep-conj-assoc*, *simp*)
**finally show** *?thesis* **by** (*simp add*: *sep-conj-com*)
**qed**

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

**lemma** *empty-empty*[*simp*]: □ *0*
**by** (*simp add*: *emp-def*)

**lemma** *sep-conj-empty*[*simp*]:
(*P* $**$ □) = *P*
**by** (*simp add*: *sep-conj-def emp-def*)

**lemma** *sep-conj-empty′*[*simp*]:
(□ $**$ *P*) = *P*
**by** (*subst sep-conj-com*, *rule sep-conj-empty*)

**lemma** *sep-conj-sep-emptyI*:
*P s* $\Longrightarrow$ (*P* $**$ □) *s*
**by** *simp*

**lemma** *sep-conj-true-P*[*simp*]:

(*sep-true* ** (*sep-true* ** *P*)) = (*sep-true* ** *P*)
**by** (*simp add*: *sep-conj-assoc*)

**lemma** *sep-conj-disj*:
((λ*s*. *P s* ∨ *Q s*) ** *R*) *s* = ((*P* ** *R*) *s* ∨ (*Q* ** *R*) *s*) (**is** *?x* = (*?y* ∨ *?z*))
**by** (*auto simp*: *sep-conj-def*)

**lemma** *sep-conj-conj*:
((λ*s*. *P s* ∧ *Q s*) ** *R*) *s* ⟹ (*P* ** *R*) *s* ∧ (*Q* ** *R*) *s*
**by** (*force intro*: *sep-conjI elim*!: *sep-conjE*)

**lemma** *sep-conj-exists1*:
((λ*s*. ∃ *x*. *P x s*) ** *Q*) *s* = (∃ *x*. (*P x* ** *Q*) *s*)
**by** (*force intro*: *sep-conjI elim*: *sep-conjE*)

**lemma** *sep-conj-exists2*:
(*P* ** (λ*s*. ∃ *x*. *Q x s*)) = (λ*s*. (∃ *x*. (*P* ** *Q x*) *s*))
**by** (*force intro*!: *sep-conjI ext elim*!: *sep-conjE*)

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

**lemma** *sep-conj-forall*:
((λ*s*. ∀ *x*. *P x s*) ** *Q*) *s* ⟹ (*P x* ** *Q*) *s*
**by** (*force intro*: *sep-conjI elim*: *sep-conjE*)

**lemma** *sep-conj-impl*:
⟦ (*P* ** *Q*) *s*; ⋀*s*. *P s* ⟹ *P*′ *s*; ⋀*s*. *Q s* ⟹ *Q*′ *s* ⟧ ⟹ (*P*′ ** *Q*′) *s*
**by** (*erule sep-conjE*, *auto intro*!: *sep-conjI*)

**lemma** *sep-conj-impl1*:
**assumes** *P*: ⋀*s*. *P s* ⟹ *I s*
**shows** (*P* ** *R*) *s* ⟹ (*I* ** *R*) *s*
**by** (*auto intro*: *sep-conj-impl P*)

**lemma** *sep-conj-sep-true-left*:
(*P* ** *Q*) *s* ⟹ (*sep-true* ** *Q*) *s*
**by** (*erule sep-conj-impl*, *simp*+)

**lemma** *sep-conj-sep-true-right*:
(*P* ** *Q*) *s* ⟹ (*P* ** *sep-true*) *s*
**by** (*subst* (*asm*) *sep-conj-com*, *drule sep-conj-sep-true-left*,
*simp add*: *sep-conj-ac*)

**lemma** *sep-globalise*:
⟦ (*P* ** *R*) *s*; (⋀*s*. *P s* ⟹ *Q s*) ⟧ ⟹ (*Q* ** *R*) *s*
**by** (*fast elim*: *sep-conj-impl*)

**lemma** *sep-implI*:
**assumes** *a*: ⋀*h*′ *h*″. ⟦ *h* ⊕ *h*′ = *Some h*″; *P h*′ ⟧ ⟹ *Q h*″

**shows** $(P \longrightarrow^* Q)\ h$
**unfolding** *sep-impl-def* **by** (*auto elim*: *a*)

**lemma** *sep-implD*:
$(x \longrightarrow^* y)\ h \Longrightarrow \forall h'\ h''.\ h \oplus h' = Some\ h'' \wedge x\ h' \longrightarrow y\ h''$
**by** (*force simp*: *sep-impl-def*)

**lemma** *sep-impl-sep-true*[*simp*]:
$(P \longrightarrow^* sep\text{-}true) = sep\text{-}true$
**by** (*force intro*!: *sep-implI ext*)

**lemma** *sep-impl-sep-false*[*simp*]:
$(sep\text{-}false \longrightarrow^* P) = sep\text{-}true$
**by** (*force intro*!: *sep-implI ext*)

**lemma** *sep-impl-sep-true-P*:
$(sep\text{-}true \longrightarrow^* P)\ s \Longrightarrow P\ s$
**apply** (*drule sep-implD*)
**apply** (*erule-tac x=0* **in** *allE*)
**apply** *simp*
**done**

**lemma** *sep-impl-sep-true-false*[*simp*]:
$(sep\text{-}true \longrightarrow^* sep\text{-}false) = sep\text{-}false$
**by** (*force intro*!: *ext dest*: *sep-impl-sep-true-P*)

**lemma** *sep-conj-sep-impl*:
$[\![\ P\ s;\ \bigwedge s.\ (P \mathbin{**} Q)\ s \Longrightarrow R\ s\ ]\!] \Longrightarrow (Q \longrightarrow^* R)\ s$
**proof** (*rule sep-implI*)
**fix** $h'\ h\ h''$
**assume** $P\ h$ **and** $h \oplus h' = Some\ h''$ **and** $Q\ h'$
**hence** $(P \mathbin{**} Q)\ h''$ **by** (*force intro*: *sep-conjI*)
**moreover assume** $\bigwedge s.\ (P \mathbin{**} Q)\ s \Longrightarrow R\ s$
**ultimately show** $R\ h''$ **by** *simp*
**qed**

**lemma** *sep-conj-sep-impl2*:
$[\![\ (P \mathbin{**} Q)\ s;\ \bigwedge s.\ P\ s \Longrightarrow (Q \longrightarrow^* R)\ s\ ]\!] \Longrightarrow R\ s$
**by** (*force dest*: *sep-implD elim*: *sep-conjE*)

**lemma** *sep-conj-sep-impl-sep-conj2*:
$(P \mathbin{**} R)\ s \Longrightarrow (P \mathbin{**} (Q \longrightarrow^* (Q \mathbin{**} R)))\ s$
**by** (*erule* (*1*) *sep-conj-impl*, *erule sep-conj-sep-impl*, *simp add*: *sep-conj-ac*)

**lemma** *sep-conj-triv-strip2*:
$Q = R \Longrightarrow (Q \mathbin{**} P) = (R \mathbin{**} P)$ **by** *simp*

**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*
**by** (*simp add*: *add2-def*)

**lemma** *add2-comm*: *add2 x y* = *add2 y x*
**by** (*auto simp*: *add2-def sep-add-commute sep-disj-commute*)

**lemma** *add2-assoc*:
*lift2 add2 a* (*lift2 add2 b c*) = *lift2 add2* (*lift2 add2 a b*) *c*
**by** (*auto simp*: *add2-def lift2-def sep-add-assoc*
*dest*: *sep-disj-addD sep-disj-addI3*
*sep-add-disjD sep-disj-addI2 sep-disj-commuteI*
*split*: *option.splits*)

**interpretation** *total-partial*: *sep-algebra-alt 0 add2*
**by** (*unfold-locales*) (*auto intro*: *add2-zero add2-comm add2-assoc*)

# 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*
**by** *simp*

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

**by** (*clarsimp simp*: *disjoint-def add-comm*)

**lemma** *sep-add-zero′*:
*sep-add′ x 0 = x*
**by** (*simp add*: *sep-add′-def*)

**lemma** *sep-add-commute′*:
*disjoint x y ⟹ sep-add′ x y = sep-add′ y x*
**by** (*clarsimp simp*: *sep-add′-def disjoint-def add-comm*)

**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*)
**using** *add-assoc* [*of Some x Some y Some z*]
**by** (*clarsimp simp*: *disjoint-def sep-add′-def lift2-def*
*split*: *option.splits*)

**lemma** *sep-disj-addD1′*:
*disjoint x* (*sep-add′ y z*) ⟹ *disjoint y z* ⟹ *disjoint x y*
**proof** (*clarsimp simp*: *disjoint-def sep-add′-def*)
**fix** *a* **assume** *a*: *y* ⊕ *z* = *Some a*
**fix** *b* **assume** *b*: *x* ⊕ *a* = *Some b*
**with** *a* **have** *Some x* ++ (*Some y* ++ *Some z*) = *Some b* **by** (*simp add*: *lift2-def*)
**hence** (*Some x* ++ *Some y*) ++ *Some z* = *Some b* **by** (*simp add*: *add-assoc*)
**thus** ∃ *b*. *x* ⊕ *y* = *Some b* **by** (*simp add*: *lift2-def split*: *option.splits*)
**qed**

**lemma** *sep-disj-addI1′*:
*disjoint x* (*sep-add′ y z*) ⟹ *disjoint y z* ⟹ *disjoint* (*sep-add′ x y*) *z*
**apply** (*clarsimp simp*: *disjoint-def sep-add′-def*)
**apply** (*rule conjI*)
**apply** *clarsimp*
**apply** (*frule lift-to-add2*, *assumption*)
**apply** (*simp add*: *add-assoc*)
**apply** (*clarsimp simp*: *lift2-def add-comm*)
**apply** *clarsimp*
**apply** (*frule lift-to-add2*, *assumption*)
**apply** (*simp add*: *add-assoc*)
**apply** (*clarsimp simp*: *lift2-def*)
**done**

**interpretation** *partial-total*: *sep-algebra sep-add′ 0 disjoint*
**apply** (*unfold-locales*)
**apply** (*rule sep-disj-zero′*)
**apply** (*erule sep-disj-commuteI′*)
**apply** (*rule sep-add-zero′*)
**apply** (*erule sep-add-commute′*)
**apply** (*erule* (*2*) *sep-add-assoc′*)
**apply** (*erule* (*1*) *sep-disj-addD1′*)

**apply** (*erule* (*1*) *sep-disj-addI1'*)
**done**

**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*
  **by** (*simp-all add*:*cap-object-def cap-objects-def cap-has-object-def*)

**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*:
$[\![x \in A;\ A \cap B = \{\}]\!] \Longrightarrow x \notin B$
**by** *fastforce*

**lemma** *union-intersection*:
$A \cap (A \cup B) = A$
$B \cap (A \cup B) = B$
$(A \cup B) \cap A = A$
$(A \cup B) \cap B = B$
**by** *fastforce+*

**lemma** *union-intersection1*: $A \cap (A \cup B) = A$
**by** (*rule inf-sup-absorb*)
**lemma** *union-intersection2*: $B \cap (A \cup B) = B$
**by** *fastforce*

**lemma** *restrict-map-disj'*:
$S \cap T = \{\} \Longrightarrow h \mid` S \perp h' \mid` T$
**by** (*auto simp*: *map-disj-def restrict-map-def dom-def*)

**lemma** *map-add-restrict-comm*:
$S \cap T = \{\} \Longrightarrow h \mid` S ++ h' \mid` T = h' \mid` T ++ h \mid` S$
**apply** (*drule restrict-map-disj'*)
**apply** (*erule map-add-com*)
**done**

**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*
**by** (*fastforce simp*: *the-set-def*)

**lemma** *the-set-inter* [*simp*]:
*the-set* (*A* ∩ *B*) = *the-set A* ∩ *the-set B*
**by** (*fastforce simp*: *the-set-def*)

**lemma** *the-set-inter-empty*:
*A* ∩ *B* = {} ⟹ *the-set A* ∩ *the-set B* = {}
**by** (*fastforce simp*: *the-set-def*)

**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*)
**apply** (*clarsimp simp*: *add-to-slots-def update-slots-def object-slots-def*)
**apply** (*fastforce simp*: *cdl-tcb.splits cdl-cnode.splits*
*split*: *cdl-object.splits*)
**done**

**lemma** *add-to-slots-twice* [*simp*]:
*add-to-slots x* (*add-to-slots y a*) = *add-to-slots* (*x* ++ *y*) *a*
**by** (*fastforce simp*: *add-to-slots-def update-slots-def object-slots-def*
*split*: *cdl-object.splits*)

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

**by** (*simp add*: *slots-of-heap-def*)

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

**lemma** *update-slots-add-to-slots-empty* [*simp*]:
*update-slots Map.empty* (*add-to-slots new obj*) = *update-slots Map.empty obj*
**by** (*clarsimp simp*: *update-slots-def add-to-slots-def split*:*cdl-object.splits*)

**lemma** *update-object-slots-id* [*simp*]: *update-slots* (*object-slots a*) *a* = *a*
**by** (*clarsimp simp*: *update-slots-def object-slots-def*
*split*: *cdl-object.splits*)

**lemma** *update-slots-of-heap-id* [*simp*]:
*h obj-id* = *Some obj* ⟹ *update-slots* (*slots-of-heap h obj-id*) *obj* = *obj*
**by** (*clarsimp simp*: *update-slots-def slots-of-heap-def object-slots-def*
*split*: *cdl-object.splits*)

**lemma** *add-to-slots-empty* [*simp*]: *add-to-slots Map.empty h* = *h*
**by** (*simp add*: *add-to-slots-def*)

**lemma** *update-slots-eq*:
*update-slots a o1* = *update-slots a o2* ⟹ *update-slots b o1* = *update-slots b o2*
**by** (*fastforce simp*: *update-slots-def cdl-tcb.splits cdl-cnode.splits*
*split*: *cdl-object.splits*)

**definition**
*not-conflicting-objects* :: *sep-state* ⇒ *sep-state* ⇒ *cdl-object-id* ⇒ *bool*
**where**
*not-conflicting-objects 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-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* ≡ 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* *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*
**apply** (*clarsimp simp*: *not-conflicting-objects-def split*:*option.splits*)

**apply** (*fastforce simp*: *update-slots-def cdl-tcb.splits cdl-cnode.splits*
*split*: *cdl-object.splits*)
**done**

**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*
**apply** (*clarsimp simp*: *object-type-def split*: *cdl-object.splits*)
**apply** (*clarsimp simp*: *object-clean-def object-clean-slots-def object-clean-fields-def*
*add-to-slots-def object-slots-def update-slots-def*
*cdl-tcb.splits cdl-cnode.splits*
*split*: *cdl-object.splits*)+
**done**

**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*
**apply** (*clarsimp simp*: *add-to-slots-def update-slots-def object-slots-def*)
**apply** (*fastforce simp*: *object-type-def cdl-tcb.splits cdl-cnode.splits*
*split*: *cdl-object.splits*)
**done**

**lemma** *not-conflicting-objects-empty* [*simp*]:
*not-conflicting-objects s* (*SepState Map.empty* (λ*obj-id*. {})) *obj-id*
**by** (*clarsimp simp*: *not-conflicting-objects-def split*:*option.splits*)

**lemma** *empty-not-conflicting-objects* [*simp*]:
*not-conflicting-objects* (*SepState Map.empty* (λ*obj-id*. {})) *s obj-id*
**by** (*clarsimp simp*: *not-conflicting-objects-def split*:*option.splits*)

**lemma** *not-conflicting-objects-empty-object* [*elim*!]:
(*sep-heap x*) *obj-id* = *None* ⟹ *not-conflicting-objects x y obj-id*
**by** (*clarsimp simp*: *not-conflicting-objects-def*)

**lemma** *empty-object-not-conflicting-objects* [*elim*!]:
(*sep-heap y*) *obj-id* = *None* ⟹ *not-conflicting-objects x y obj-id*
**apply** (*drule not-conflicting-objects-empty-object* [**where** *y*=*x*])
**apply** (*clarsimp simp*: *not-conflicting-objects-comm*)
**done**

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

**lemma** *empty-cdl-heap-add* [*simp*]:

*cdl-heap-add* (*SepState Map.empty* (λ*obj-id*. {})) (*SepState h gs*)= *h*
**apply** (*simp add*: *cdl-heap-add-def*)
**apply** (*rule ext*)
**apply** (*clarsimp split*: *option.splits*)
**done**

**lemma** *map-add-result-empty1*: *a* ++ *b* = *Map.empty* ⟹ *a* = *Map.empty*
**apply** (*subgoal-tac dom* (*a*++*b*) = {})
**apply** (*subgoal-tac dom* (*a*) = {})
**apply** *clarsimp*
**apply** (*unfold dom-map-add*)[*1*]
**apply** *clarsimp*
**apply** *clarsimp*
**done**

**lemma** *map-add-result-empty2*: *a* ++ *b* = *Map.empty* ⟹ *b* = *Map.empty*
**apply** (*subgoal-tac dom* (*a*++*b*) = {})
**apply** (*subgoal-tac dom* (*a*) = {})
**apply** *clarsimp*
**apply** (*unfold dom-map-add*)[*1*]
**apply** *clarsimp*
**apply** *clarsimp*
**done**

**lemma** *map-add-emptyE* [*elim*!]: ⟦*a* ++ *b* = *Map.empty*; ⟦*a* = *Map.empty*; *b* = *Map.empty*⟧ ⟹ *R*⟧ ⟹ *R*
**apply** (*frule map-add-result-empty1*)
**apply** (*frule map-add-result-empty2*)
**apply** *clarsimp*
**done**

**lemma** *clean-slots-empty* [*simp*]:
*clean-slots Map.empty cmp* = *Map.empty*
**by** (*clarsimp simp*: *clean-slots-def*)

**lemma** *object-type-update-slots* [*simp*]:
*object-type* (*update-slots slots x*) = *object-type x*
**by** (*clarsimp simp*: *object-type-def update-slots-def split*: *cdl-object.splits*)

**lemma** *object-type-object-clean-slots* [*simp*]:
*object-type* (*object-clean-slots x cmp*) = *object-type x*
**by** (*clarsimp simp*: *object-clean-slots-def*)

**lemma** *object-type-object-clean-fields* [*simp*]:
*object-type* (*object-clean-fields x cmp*) = *object-type x*
**by** (*clarsimp simp*: *object-clean-fields-def object-type-def split*: *cdl-object.splits*)

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

**by** (*clarsimp simp*: *object-clean-def*)

**lemma** *object-type-add-to-slots* [*simp*]:
*object-type* (*add-to-slots slots x*) = *object-type x*
**by** (*clarsimp simp*: *object-type-def add-to-slots-def update-slots-def split*: *cdl-object.splits*)

**lemma** *object-slots-update-slots* [*simp*]:
*has-slots obj* ⟹ *object-slots* (*update-slots slots obj*) = *slots*
**by** (*clarsimp simp*: *object-slots-def update-slots-def has-slots-def*
*split*: *cdl-object.splits*)

**lemma** *object-slots-update-slots-empty* [*simp*]:
¬*has-slots obj* ⟹ *object-slots* (*update-slots slots obj*) = *Map.empty*
**by** (*clarsimp simp*: *object-slots-def update-slots-def has-slots-def*
*split*: *cdl-object.splits*)

**lemma** *update-slots-no-slots* [*simp*]:
¬*has-slots obj* ⟹ *update-slots slots obj* = *obj*
**by** (*clarsimp simp*: *update-slots-def has-slots-def split*: *cdl-object.splits*)

**lemma** *update-slots-update-slots* [*simp*]:
*update-slots slots* (*update-slots slots′ obj*) = *update-slots slots obj*
**by** (*clarsimp simp*: *update-slots-def split*: *cdl-object.splits*)

**lemma** *update-slots-same-object*:
*a* = *b* ⟹ *update-slots a obj* = *update-slots b obj*
**by** (*erule arg-cong*)

**lemma** *object-type-has-slots*:
⟦*has-slots x*; *object-type x* = *object-type y*⟧ ⟹ *has-slots y*
**by** (*clarsimp simp*: *object-type-def has-slots-def split*: *cdl-object.splits*)

**lemma** *object-slots-object-clean-fields* [*simp*]:
*object-slots* (*object-clean-fields obj cmp*) = *object-slots obj*
**by** (*clarsimp simp*: *object-slots-def object-clean-fields-def split*: *cdl-object.splits*)

**lemma** *object-slots-object-clean-slots* [*simp*]:
*object-slots* (*object-clean-slots obj cmp*) = *clean-slots* (*object-slots obj*) *cmp*
**by** (*clarsimp simp*: *object-clean-slots-def object-slots-def update-slots-def split*:
*cdl-object.splits*)

**lemma** *object-slots-object-clean* [*simp*]:
*object-slots* (*object-clean obj cmp*) = *clean-slots* (*object-slots obj*) *cmp*
**by** (*clarsimp simp*: *object-clean-def*)

**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*
**by** (*clarsimp simp*: *object-slots-def add-to-slots-def update-slots-def object-type-def*

*split*: *cdl-object.splits*)

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

**lemma** *object-clean-fields-idem* [*simp*]:
*object-clean-fields* (*object-clean-fields obj cmp*) *cmp* = *object-clean-fields obj cmp*
**by** (*clarsimp simp*: *object-clean-fields-def split*: *cdl-object.splits*)

**lemma** *object-clean-slots-idem* [*simp*]:
*object-clean-slots* (*object-clean-slots obj cmp*) *cmp* = *object-clean-slots obj cmp*
**apply** (*case-tac has-slots obj*)
**apply** (*clarsimp simp*: *object-clean-slots-def clean-slots-def*)+
**done**

**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*
**by** (*clarsimp simp*: *object-clean-fields-def object-clean-slots-def clean-slots-def object-slots-def update-slots-def split*: *cdl-object.splits*)

**lemma** *object-clean-idem* [*simp*]:
*object-clean* (*object-clean obj cmp*) *cmp* = *object-clean obj cmp*
**by** (*clarsimp simp*: *object-clean-def*)

**lemma** *has-slots-object-clean-slots*:
*has-slots* (*object-clean-slots obj cmp*) = *has-slots obj*
**by** (*clarsimp simp*: *has-slots-def object-clean-slots-def update-slots-def split*: *cdl-object.splits*)

**lemma** *has-slots-object-clean-fields*:
*has-slots* (*object-clean-fields obj cmp*) = *has-slots obj*
**by** (*clarsimp simp*: *has-slots-def object-clean-fields-def split*: *cdl-object.splits*)

**lemma** *has-slots-object-clean*:
*has-slots* (*object-clean obj cmp*) = *has-slots obj*
**by** (*clarsimp simp*: *object-clean-def has-slots-object-clean-slots has-slots-object-clean-fields*)

**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*)
**apply** (*case-tac has-slots obj*)
**apply** (*clarsimp simp*: *has-slots-object-clean-fields*)+
**done**

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

**by** (*clarsimp simp*: *object-clean-fields-def update-slots-def split*: *cdl-object.splits*)

**lemma** *object-clean-fields-twice* [*simp*]:
(*object-clean-fields* (*object-clean-fields obj cmp'*) *cmp*) = *object-clean-fields obj* (*cmp* ∩ *cmp'*)
**by** (*clarsimp simp*: *object-clean-fields-def split*: *cdl-object.splits*)

**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'*)
**by** (*fastforce simp*: *update-slots-def object-clean-fields-def object-type-def split*: *cdl-object.splits*)

**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'*
**by** (*fastforce simp*: *object-clean-fields-def object-type-def has-slots-def split*: *cdl-object.splits*)

**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'*)
**apply** (*clarsimp simp*: *object-clean-def object-clean-slots-def*)
**apply** (*erule* (*2*) *update-slots-object-clean-fields*)
**done**

**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*))
**apply** (*rule ext*)
**apply** (*rename-tac obj-id*)
**apply** (*erule-tac x*=*obj-id* **in** *allE*)
**apply** (*clarsimp simp*: *cdl-heap-add-def cdl-ghost-state-add-def not-conflicting-objects-def*)
**apply** (*simp add*: *Let-unfold split*: *option.splits*)
**apply** (*rename-tac obj-y obj-x obj-z*)
**apply** (*clarsimp simp*: *object-add-def clean-slots-def object-clean-def object-clean-slots-def Let-unfold*)
**apply** (*case-tac has-slots obj-z*)
**apply** (*subgoal-tac has-slots obj-y*)
**apply** (*subgoal-tac has-slots obj-x*)
**apply** ((*clarsimp simp*: *has-slots-object-clean-fields has-slots-object-clean-slots has-slots-object-clean*
*map-add-restrict union-intersection* |
*drule inter-empty-not-both* |

*erule update-slots-object-clean-fields* |
*erule object-type-has-slots*, *simp* |
*simp* | *safe*)+)[*3*]
**apply** (*subgoal-tac* ¬ *has-slots obj-y*)
**apply** (*subgoal-tac* ¬ *has-slots obj-x*)
**apply** ((*clarsimp simp*: *has-slots-object-clean-fields has-slots-object-clean-slots has-slots-object-clean*
*map-add-restrict union-intersection* |
*drule inter-empty-not-both* |
*erule object-clean-fields-no-slots* |
*erule object-type-has-slots*, *simp* |
*simp* | *safe*)+)
**apply** (*fastforce simp*: *object-type-has-slots*)+
**done**

**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*))
**apply** (*clarsimp simp*: *sep-state-disj-def*)
**apply** (*cut-tac cdl-heap-add-assoc'*)
**apply** *fast*
**apply** *fastforce*
**done**

**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*))
**apply** (*rule ext*)
**apply** (*fastforce simp*: *cdl-heap-add-def cdl-ghost-state-add-def Let-unfold*)
**done**

**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*
**apply** (*clarsimp simp*: *clean-slots-def*)
**apply** (*drule the-set-inter-empty*)
**apply** (*erule map-add-restrict-comm*)
**done**

**lemma** *object-clean-all*:
*object-type obj-a* = *object-type obj-b* ⟹ *object-clean obj-b* {} = *object-clean obj-a* {}
**apply** (*clarsimp simp*: *object-clean-def object-clean-slots-def clean-slots-def the-set-def*)
**apply** (*rule-tac cmps'1*={} **and** *obj'1*=*obj-a* **in** *trans* [*OF update-slots-object-clean-fields*], *fastforce*+)
**done**

**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*
**apply** (*clarsimp simp*: *object-add-def Let-unfold*)
**apply** (*rule conjI* | *clarsimp*)+
**apply** *fastforce*
**apply** (*rule conjI* | *clarsimp*)+
**apply** (*drule-tac slots-a = object-slots obj-a* **and** *slots-b = object-slots obj-b* **in** *clean-slots-map-add-comm*)
**apply** *fastforce*
**apply** (*rule conjI* | *clarsimp*)+
**apply** (*drule-tac slots-a = object-slots obj-a* **and** *slots-b = object-slots obj-b* **in** *clean-slots-map-add-comm*)
**apply** *fastforce*
**apply** (*rule conjI* | *clarsimp*)+
**apply** (*erule object-clean-all*)
**apply** (*clarsimp*)
**apply** (*rule-tac cmps'1=cmps-b* **and** *obj'1=obj-b* **in** *trans* [*OF update-slots-object-clean*], *assumption*+)
**apply** (*drule-tac slots-a = object-slots obj-a* **and** *slots-b = object-slots obj-b* **in** *clean-slots-map-add-comm*)
**apply** *fastforce*
**done**

**lemma** *sep-state-add-comm*:
*sep-state-disj x y* ⟹ *sep-state-add x y = sep-state-add y x*
**apply** (*clarsimp simp*: *sep-state-add-def sep-state-disj-def*)
**apply** (*rule conjI*)
**apply** (*case-tac x*, *case-tac y*, *clarsimp*)
**apply** (*rename-tac heap-a gs-a heap-b gs-b*)
**apply** (*clarsimp simp*: *cdl-heap-add-def Let-unfold*)
**apply** (*rule ext*)
**apply** (*case-tac heap-a obj-id*)
**apply** (*case-tac heap-b obj-id*, *simp-all add*: *slots-of-heap-def*)
**apply** (*case-tac heap-b obj-id*, *simp-all add*: *slots-of-heap-def*)
**apply** (*rename-tac obj-a obj-b*)
**apply** (*erule-tac x=obj-id* **in** *allE*)
**apply** (*rule object-add-comm*)
**apply** (*clarsimp simp*: *not-conflicting-objects-def*)
**apply** (*clarsimp simp*: *not-conflicting-objects-def*)
**apply** (*rule ext*, *fastforce simp*: *cdl-ghost-state-add-def Let-unfold Un-commute*)
**done**

**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*
**by** (*fastforce simp*: *add-to-slots-def update-slots-def object-slots-def*
*cdl-tcb.splits cdl-cnode.splits*

*dest*!: *map-add-com*
*split*: *cdl-object.splits*)

**lemma** *cdl-heap-add-none1*:
*cdl-heap-add x y obj-id = None* ⟹ (*sep-heap x*) *obj-id = None*
**by** (*clarsimp simp*: *cdl-heap-add-def Let-unfold split*:*option.splits if-split-asm*)

**lemma** *cdl-heap-add-none2*:
*cdl-heap-add x y obj-id = None* ⟹ (*sep-heap y*) *obj-id = None*
**by** (*clarsimp simp*: *cdl-heap-add-def Let-unfold split*:*option.splits if-split-asm*)

**lemma** *object-type-object-addL*:
*object-type obj = object-type obj'*
⟹ *object-type* (*object-add obj obj' cmp cmp'*) = *object-type obj*
**by** (*clarsimp simp*: *object-add-def Let-unfold*)

**lemma** *object-type-object-addR*:
*object-type obj = object-type obj'*
⟹ *object-type* (*object-add obj obj' cmp cmp'*) = *object-type obj'*
**by** (*clarsimp simp*: *object-add-def Let-unfold*)

**lemma** *sep-state-add-disjL*:
⟦*sep-state-disj y z*; *sep-state-disj x* (*sep-state-add y z*)⟧ ⟹ *sep-state-disj x y*
**apply** (*clarsimp simp*: *sep-state-disj-def sep-state-add-def*)
**apply** (*rename-tac obj-id*)
**apply** (*clarsimp simp*: *not-conflicting-objects-def*)
**apply** (*erule-tac x*=*obj-id* **in** *allE*)+
**apply** (*fastforce simp*: *cdl-heap-add-def cdl-ghost-state-add-def object-type-object-addR*
*split*: *option.splits*)
**done**

**lemma** *sep-state-add-disjR*:
⟦*sep-state-disj y z*; *sep-state-disj x* (*sep-state-add y z*)⟧ ⟹ *sep-state-disj x z*
**apply** (*clarsimp simp*: *sep-state-disj-def sep-state-add-def*)
**apply** (*rename-tac obj-id*)
**apply** (*clarsimp simp*: *not-conflicting-objects-def*)
**apply** (*erule-tac x*=*obj-id* **in** *allE*)+
**apply** (*fastforce simp*: *cdl-heap-add-def cdl-ghost-state-add-def object-type-object-addR*
*split*: *option.splits*)
**done**

**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*)
**apply** (*clarsimp simp*: *sep-state-disj-def sep-state-add-def*)
**apply** (*rename-tac obj-id*)
**apply** (*clarsimp simp*: *not-conflicting-objects-def*)
**apply** (*erule-tac x*=*obj-id* **in** *allE*)+
**apply** (*fastforce simp*: *cdl-heap-add-def cdl-ghost-state-add-def object-type-object-addR*

*split*: *option.splits*)
**done**

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

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

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

**instance**
**apply** *standard*

**apply** (*simp add*: *sep-disj-sep-state-def sep-state-disj-def zero-sep-state-def*)

**apply** (*clarsimp simp*: *not-conflicting-objects-comm sep-disj-sep-state-def sep-state-disj-def Let-unfold*
*map-disj-com not-conflicting-objects-comm Int-commute*)

**apply** (*simp add*: *plus-sep-state-def sep-state-add-def zero-sep-state-def*)
**apply** (*case-tac x*)
**apply** (*clarsimp simp*: *cdl-heap-add-def*)
**apply** (*rule ext*)
**apply** (*clarsimp simp*: *cdl-ghost-state-add-def split*:*if-split-asm*)

**apply** (*clarsimp simp*: *plus-sep-state-def sep-disj-sep-state-def*)
**apply** (*erule sep-state-add-comm*)

**apply** (*simp add*: *plus-sep-state-def sep-state-add-def*)
**apply** (*rule conjI*)
**apply** (*clarsimp simp*: *sep-disj-sep-state-def*)
**apply** (*erule* (*2*) *cdl-heap-add-assoc*)
**apply** (*rule cdl-ghost-state-add-assoc*)

**apply** (*clarsimp simp*: *plus-sep-state-def sep-disj-sep-state-def*)
**apply** (*rule iffI*)

**apply** (*rule conjI*)

**apply** (*erule* (*1*) *sep-state-add-disjL*)

**apply** (*erule* (*1*) *sep-state-add-disjR*)

**apply** *clarsimp*
**apply** (*erule* (*2*) *sep-state-add-disj*)
**done**

**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)
 apply (clarsimp simp: sep-map-general-def object-at-def)
 apply (rule)
  apply clarsimp
 apply (clarsimp simp: fun-upd-def)
 apply (rule ext)
 apply (fastforce simp: dom-def split:if-split)
 done


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 ≡ λs. 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 ≡ (λp 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.