

Separation Algebra

Gerwin Klein and Rafal Kolanski and Andrew Boyton

April 10, 2026

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	3
2	Input syntax for lifting boolean predicates to separation predicates	3
3	Associative/Commutative Monoid Basis of Separation Algebras	4
4	Separation Algebra as Defined by Calcagno et al.	4
4.1	Basic Construct Definitions and Abbreviations	5
4.2	Disjunction/Addition Properties	5
4.3	Substate Properties	6
4.4	Separating Conjunction Properties	6
4.5	Properties of <i>sep-true</i> and <i>sep-false</i>	7
4.6	Properties of zero (\square)	8
4.7	Properties of top (<i>sep-true</i>)	8
4.8	Separating Conjunction with Quantifiers	9
4.9	Properties of Separating Implication	9

4.10	Pure assertions	10
4.11	Intuitionistic assertions	11
4.12	Strictly exact assertions	13
5	Separation Algebra with Stronger, but More Intuitive Disjunction Axiom	13
6	Folding separating conjunction over lists of predicates	14
7	Separation Algebra with a Cancellative Monoid (for completeness)	14
8	Standard Heaps as an Instance of Separation Algebra	15
9	Separation Logic Tactics	16
10	Selection (move-to-front) tactics	16
11	Substitution	17
12	Forward Reasoning	17
13	Backward Reasoning	17
14	Cancellation of Common Conjuncts via Elimination Rules	17
15	Example from HOL/Hoare/Separation	17
16	Test cases for <i>sep-cancel</i>.	20
17	More properties of maps plus map disjunction.	21
18	Things that could go into Option Type	21
19	Things that go into Map.thy	21
19.1	Properties of maps not related to restriction	22
19.2	Properties of map restriction	23
20	Things that should not go into Map.thy (separation logic)	25
20.1	Definitions	25
20.2	Properties of ($'-$)	25
20.3	Properties of map disjunction	25
20.4	Map associativity-commutativity based on map disjunction . .	25
20.5	Basic properties	26
20.6	Map disjunction and addition	27
20.7	Map disjunction and map updates	28
20.8	Map disjunction and (\subseteq_m)	28

20.9 Map disjunction and restriction	29
21 Separation Algebra for Virtual Memory	30
22 Abstract Separation Logic, Alternative Definition	32
23 Equivalence between Separation Algebra Formulations	37
24 Total implies Partial	38
25 Partial implies Total	38
26 A simplified version of the actual capDL specification.	39
27 Instantiating capDL as a separation algebra.	43
28 Defining some separation logic maps-to predicates on top of the instantiation.	53

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 ⇒ bool) ⇒ ('a ⇒ bool) ⇒ 'a ⇒ bool (infixr ‹and› 35) where
  a and b ≡ λs. a s ∧ b s
```

```
abbreviation (input)
  pred-or :: ('a ⇒ bool) ⇒ ('a ⇒ bool) ⇒ 'a ⇒ bool (infixr ‹or› 30) where
  a or b ≡ λs. a s ∨ b s
```

```
abbreviation (input)
  pred-not :: ('a ⇒ bool) ⇒ 'a ⇒ bool (‹not -> [40] 40) where
  not a ≡ λs. ¬a s
```

```
abbreviation (input)
  pred-imp :: ('a ⇒ bool) ⇒ ('a ⇒ bool) ⇒ 'a ⇒ bool (infixr ‹imp› 25) where
  a imp b ≡ λs. a s ⟶ b s
```

```
abbreviation (input)
```

pred-K :: 'b ⇒ 'a ⇒ 'b (⟨⟨-⟩⟩) **where**
 ⟨f⟩ ≡ λs. f

abbreviation (*input*)

pred-ex :: ('b ⇒ 'a ⇒ bool) ⇒ 'a ⇒ bool (**binder** ⟨EXS⟩ 10) **where**
 EXS x. P x ≡ λs. ∃x. P x s

abbreviation (*input*)

pred-all :: ('b ⇒ 'a ⇒ bool) ⇒ 'a ⇒ bool (**binder** ⟨ALLS⟩ 10) **where**
 ALLS x. P x ≡ λs. ∀x. P x s

3 Associative/Commutative Monoid Basis of Separation Algebras

class *pre-sep-algebra* = zero + plus +

fixes *sep-disj* :: 'a => 'a => bool (**infix** ⟨##⟩ 60)

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

assumes *sep-disj-commuteI*: x ## y ⇒ y ## x

assumes *sep-add-zero* [*simp*]: x + 0 = x

assumes *sep-add-commute*: x ## y ⇒ x + y = y + x

assumes *sep-add-assoc*:

[[x ## y; y ## z; x ## z]] ⇒ (x + y) + z = x + (y + z)

begin

lemma *sep-disj-commute*: x ## y = y ## x

⟨*proof*⟩

lemma *sep-add-left-commute*:

assumes a: a ## b b ## c a ## c

shows b + (a + c) = a + (b + c) (**is** ?lhs = ?rhs)

⟨*proof*⟩

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

end

4 Separation Algebra as Defined by Calcagno et al.

class *sep-algebra* = *pre-sep-algebra* +

assumes *sep-disj-addD1*: [[x ## y + z; y ## z]] ⇒ x ## y

assumes *sep-disj-addI1*: [[x ## y + z; y ## z]] ⇒ x + y ## z

begin

4.1 Basic Construct Definitions and Abbreviations

definition

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

notation

$sep\text{-}conj$ (**infixr** $\langle \wedge * \rangle$ 35)

definition

$sep\text{-}empty :: 'a \Rightarrow bool$ ($\langle \square \rangle$) **where**
 $\square \equiv \lambda h. h = 0$

definition

$sep\text{-}impl :: ('a \Rightarrow bool) \Rightarrow ('a \Rightarrow bool) \Rightarrow ('a \Rightarrow bool)$ (**infixr** $\langle \longrightarrow * \rangle$ 25)
where
 $P \longrightarrow * Q \equiv \lambda h. \forall h'. h \#\# h' \wedge P h' \longrightarrow Q (h + h')$

definition

$sep\text{-}substate :: 'a \Rightarrow 'a \Rightarrow bool$ (**infix** $\langle \preceq \rangle$ 60) **where**
 $x \preceq y \equiv \exists z. x \#\# z \wedge x + z = y$

abbreviation

$sep\text{-}true \equiv \langle True \rangle$

abbreviation

$sep\text{-}false \equiv \langle False \rangle$

definition

$sep\text{-}list\text{-}conj :: ('a \Rightarrow bool) list \Rightarrow ('a \Rightarrow bool)$ ($\langle \wedge * \rightarrow [60] 90 \rangle$) **where**
 $sep\text{-}list\text{-}conj Ps \equiv foldl (**) \square Ps$

4.2 Disjunction/Addition Properties

lemma *disjoint-zero-sym* [*simp*]: $0 \#\# x$
 $\langle proof \rangle$

lemma *sep-add-zero-sym* [*simp*]: $0 + x = x$
 $\langle proof \rangle$

lemma *sep-disj-addD2*: $\llbracket x \#\# y + z; y \#\# z \rrbracket \Longrightarrow x \#\# z$
 $\langle proof \rangle$

lemma *sep-disj-addD*: $\llbracket x \#\# y + z; y \#\# z \rrbracket \Longrightarrow x \#\# y \wedge x \#\# z$
 $\langle proof \rangle$

lemma *sep-add-disjD*: $\llbracket x + y \#\# z; x \#\# y \rrbracket \Longrightarrow x \#\# z \wedge y \#\# z$
 $\langle proof \rangle$

lemma *sep-disj-addI2*:

$$\llbracket x \#\# y + z; y \#\# z \rrbracket \Longrightarrow x + z \#\# y$$

<proof>

lemma *sep-add-disjI1*:

$$\llbracket x + y \#\# z; x \#\# y \rrbracket \Longrightarrow x + z \#\# y$$

<proof>

lemma *sep-add-disjI2*:

$$\llbracket x + y \#\# z; x \#\# y \rrbracket \Longrightarrow z + y \#\# x$$

<proof>

lemma *sep-disj-addI3*:

$$x + y \#\# z \Longrightarrow x \#\# y \Longrightarrow x \#\# y + z$$

<proof>

lemma *sep-disj-add*:

$$\llbracket y \#\# z; x \#\# y \rrbracket \Longrightarrow x \#\# y + z = x + y \#\# z$$

<proof>

4.3 Substate Properties

lemma *sep-substate-disj-add*:

$$x \#\# y \Longrightarrow x \preceq x + y$$

<proof>

lemma *sep-substate-disj-add'*:

$$x \#\# y \Longrightarrow x \preceq y + x$$

<proof>

4.4 Separating Conjunction Properties

lemma *sep-conjD*:

$$(P \wedge^* Q) h \Longrightarrow \exists x y. x \#\# y \wedge h = x + y \wedge P x \wedge Q y$$

<proof>

lemma *sep-conjE*:

$$\llbracket (P ** Q) h; \bigwedge x y. \llbracket P x; Q y; x \#\# y; h = x + y \rrbracket \Longrightarrow X \rrbracket \Longrightarrow X$$

<proof>

lemma *sep-conjI*:

$$\llbracket P x; Q y; x \#\# y; h = x + y \rrbracket \Longrightarrow (P ** Q) h$$

<proof>

lemma *sep-conj-commuteI*:

$$(P ** Q) h \Longrightarrow (Q ** P) h$$

<proof>

lemma *sep-conj-commute*:

$(P ** Q) = (Q ** P)$
 $\langle proof \rangle$

lemma *sep-conj-assoc*:
 $((P ** Q) ** R) = (P ** Q ** R)$ (**is** ?lhs = ?rhs)
 $\langle proof \rangle$

lemma *sep-conj-impl*:
 $\llbracket (P ** Q) h; \bigwedge h. P h \implies P' h; \bigwedge h. Q h \implies Q' h \rrbracket \implies (P' ** Q') h$
 $\langle proof \rangle$

lemma *sep-conj-impl1*:
assumes $P: \bigwedge h. P h \implies I h$
shows $(P ** R) h \implies (I ** R) h$
 $\langle proof \rangle$

lemma *sep-globalise*:
 $\llbracket (P ** R) h; (\bigwedge h. P h \implies Q h) \rrbracket \implies (Q ** R) h$
 $\langle proof \rangle$

lemma *sep-conj-trivial-strip2*:
 $Q = R \implies (Q ** P) = (R ** P)$ $\langle proof \rangle$

lemma *disjoint-subheaps-exist*:
 $\exists x y. x \#\# y \wedge h = x + y$
 $\langle proof \rangle$

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

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

lemma *ab-semigroup-mult-sep-conj*: *class.ab-semigroup-mult* (**)
 $\langle proof \rangle$

lemma *sep-empty-zero* [*simp,intro!*]: $\square 0$
 $\langle proof \rangle$

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

lemma *sep-conj-sep-true*:
 $P h \implies (P ** sep-true) h$
 $\langle proof \rangle$

lemma *sep-conj-sep-true'*:
 $P h \implies (sep-true ** P) h$
 $\langle proof \rangle$

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

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

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

4.6 Properties of zero (\square)

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

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

lemma *sep-conj-sep-emptyI*:
 $P h \implies (P ** \square) h$
 $\langle proof \rangle$

lemma *sep-conj-sep-emptyE*:
 $\llbracket P s; (P ** \square) s \implies (Q ** R) s \rrbracket \implies (Q ** R) s$
 $\langle proof \rangle$

lemma *monoid-add*: *class.monoid-add* ((******)) \square
 $\langle proof \rangle$

lemma *comm-monoid-add*: *class.comm-monoid-add* (******) \square
 $\langle proof \rangle$

4.7 Properties of top (*sep-true*)

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

lemma *sep-conj-disj*:
 $((P \text{ or } Q) ** R) = ((P ** R) \text{ or } (Q ** R))$
 $\langle proof \rangle$

lemma *sep-conj-sep-true-left*:
 $(P ** Q) h \implies (sep\text{-true} ** Q) h$
 $\langle proof \rangle$

lemma *sep-conj-sep-true-right*:
 $(P ** Q) h \implies (P ** \text{sep-true}) h$
 $\langle \text{proof} \rangle$

4.8 Separating Conjunction with Quantifiers

lemma *sep-conj-conj*:
 $((P \text{ and } Q) ** R) h \implies ((P ** R) \text{ and } (Q ** R)) h$
 $\langle \text{proof} \rangle$

lemma *sep-conj-exists1*:
 $((\text{EXS } x. P x) ** Q) = (\text{EXS } x. (P x ** Q))$
 $\langle \text{proof} \rangle$

lemma *sep-conj-exists2*:
 $(P ** (\text{EXS } x. Q x)) = (\text{EXS } x. P ** Q x)$
 $\langle \text{proof} \rangle$

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

lemma *sep-conj-spec*:
 $((\text{ALLS } x. P x) ** Q) h \implies (P x ** Q) h$
 $\langle \text{proof} \rangle$

4.9 Properties of Separating Implication

lemma *sep-implI*:
assumes $a: \bigwedge h'. \llbracket h \#\# h'; P h' \rrbracket \implies Q (h + h')$
shows $(P \longrightarrow* Q) h$
 $\langle \text{proof} \rangle$

lemma *sep-implD*:
 $(x \longrightarrow* y) h \implies \forall h'. h \#\# h' \wedge x h' \longrightarrow y (h + h')$
 $\langle \text{proof} \rangle$

lemma *sep-implE*:
 $(x \longrightarrow* y) h \implies (\forall h'. h \#\# h' \wedge x h' \longrightarrow y (h + h') \implies Q) \implies Q$
 $\langle \text{proof} \rangle$

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

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

lemma *sep-impl-sep-true-P*:
 $(\text{sep-true} \longrightarrow* P) h \implies P h$
 $\langle \text{proof} \rangle$

lemma *sep-impl-sep-true-false* [*simp*]:
 $(sep\ true \longrightarrow^* sep\ false) = sep\ false$
 $\langle proof \rangle$

lemma *sep-conj-sep-impl*:
 $\llbracket P\ h; \bigwedge h. (P\ **\ Q)\ h \implies R\ h \rrbracket \implies (Q \longrightarrow^* R)\ h$
 $\langle proof \rangle$

lemma *sep-conj-sep-impl2*:
 $\llbracket (P\ **\ Q)\ h; \bigwedge h. P\ h \implies (Q \longrightarrow^* R)\ h \rrbracket \implies R\ h$
 $\langle proof \rangle$

lemma *sep-conj-sep-impl-sep-conj2*:
 $(P\ **\ R)\ h \implies (P\ **\ (Q \longrightarrow^* (Q\ **\ R)))\ h$
 $\langle proof \rangle$

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*:
 $pure\ sep\ true$
 $\langle proof \rangle$

lemma *pure-sep-false*:
 $pure\ sep\ false$
 $\langle proof \rangle$

lemma *pure-split*:
 $pure\ P = (P = sep\ true \vee P = sep\ false)$
 $\langle proof \rangle$

lemma *pure-sep-conj*:
 $\llbracket pure\ P; pure\ Q \rrbracket \implies pure\ (P\ \wedge^*\ Q)$
 $\langle proof \rangle$

lemma *pure-sep-impl*:
 $\llbracket pure\ P; pure\ Q \rrbracket \implies pure\ (P \longrightarrow^* Q)$
 $\langle proof \rangle$

lemma *pure-conj-sep-conj*:
 $\llbracket (P\ and\ Q)\ h; pure\ P \vee pure\ Q \rrbracket \implies (P\ \wedge^*\ Q)\ h$
 $\langle proof \rangle$

lemma *pure-sep-conj-conj*:
 $\llbracket (P\ \wedge^*\ Q)\ h; pure\ P; pure\ Q \rrbracket \implies (P\ and\ Q)\ h$

$\langle \text{proof} \rangle$

lemma *pure-conj-sep-conj-assoc*:

$\text{pure } P \implies ((P \text{ and } Q) \wedge^* R) = (P \text{ and } (Q \wedge^* R))$

$\langle \text{proof} \rangle$

lemma *pure-sep-impl-impl*:

$\llbracket (P \longrightarrow^* Q) \text{ h}; \text{pure } P \rrbracket \implies P \text{ h} \longrightarrow Q \text{ h}$

$\langle \text{proof} \rangle$

lemma *pure-impl-sep-impl*:

$\llbracket P \text{ h} \longrightarrow Q \text{ h}; \text{pure } P; \text{pure } Q \rrbracket \implies (P \longrightarrow^* Q) \text{ h}$

$\langle \text{proof} \rangle$

lemma *pure-conj-right*: $(Q \wedge^* (\langle P' \rangle \text{ and } Q')) = (\langle P' \rangle \text{ and } (Q \wedge^* Q'))$

$\langle \text{proof} \rangle$

lemma *pure-conj-right'*: $(Q \wedge^* (P' \text{ and } \langle Q' \rangle)) = (\langle Q' \rangle \text{ and } (Q \wedge^* P'))$

$\langle \text{proof} \rangle$

lemma *pure-conj-left*: $((\langle P' \rangle \text{ and } Q') \wedge^* Q) = (\langle P' \rangle \text{ and } (Q' \wedge^* Q))$

$\langle \text{proof} \rangle$

lemma *pure-conj-left'*: $((P' \text{ and } \langle Q' \rangle) \wedge^* Q) = (\langle Q' \rangle \text{ and } (P' \wedge^* Q))$

$\langle \text{proof} \rangle$

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

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

4.11 Intuitionistic assertions

definition *intuitionistic* :: $(a \Rightarrow \text{bool}) \Rightarrow \text{bool}$ **where**

intuitionistic $P \equiv \forall h h'. P \text{ h} \wedge h \preceq h' \longrightarrow P \text{ h}'$

lemma *intuitionisticI*:

$(\bigwedge h h'. \llbracket P \text{ h}; h \preceq h' \rrbracket \implies P \text{ h}') \implies \text{intuitionistic } P$

$\langle \text{proof} \rangle$

lemma *intuitionisticD*:

$\llbracket \text{intuitionistic } P; P \text{ h}; h \preceq h' \rrbracket \implies P \text{ h}'$

$\langle \text{proof} \rangle$

lemma *pure-intuitionistic*:

$\text{pure } P \implies \text{intuitionistic } P$

$\langle \text{proof} \rangle$

lemma *intuitionistic-conj*:

$\llbracket \text{intuitionistic } P; \text{intuitionistic } Q \rrbracket \Longrightarrow \text{intuitionistic } (P \text{ and } Q)$
 $\langle \text{proof} \rangle$

lemma *intuitionistic-disj*:

$\llbracket \text{intuitionistic } P; \text{intuitionistic } Q \rrbracket \Longrightarrow \text{intuitionistic } (P \text{ or } Q)$
 $\langle \text{proof} \rangle$

lemma *intuitionistic-forall*:

$(\bigwedge x. \text{intuitionistic } (P x)) \Longrightarrow \text{intuitionistic } (\text{ALLS } x. P x)$
 $\langle \text{proof} \rangle$

lemma *intuitionistic-exists*:

$(\bigwedge x. \text{intuitionistic } (P x)) \Longrightarrow \text{intuitionistic } (\text{EXS } x. P x)$
 $\langle \text{proof} \rangle$

lemma *intuitionistic-sep-conj-sep-true*:

$\text{intuitionistic } (\text{sep-true } \wedge^* P)$
 $\langle \text{proof} \rangle$

lemma *intuitionistic-sep-impl-sep-true*:

$\text{intuitionistic } (\text{sep-true } \longrightarrow^* P)$
 $\langle \text{proof} \rangle$

lemma *intuitionistic-sep-conj*:

assumes *ip*: $\text{intuitionistic } (P :: ('a \Rightarrow \text{bool}))$
shows $\text{intuitionistic } (P \wedge^* Q)$
 $\langle \text{proof} \rangle$

lemma *intuitionistic-sep-impl*:

assumes *iq*: $\text{intuitionistic } Q$
shows $\text{intuitionistic } (P \longrightarrow^* Q)$
 $\langle \text{proof} \rangle$

lemma *strongest-intuitionistic*:

$\neg (\exists Q. (\forall h. (Q h \longrightarrow (P \wedge^* \text{sep-true}) h)) \wedge \text{intuitionistic } Q \wedge$
 $Q \neq (P \wedge^* \text{sep-true}) \wedge (\forall h. P h \longrightarrow Q h))$
 $\langle \text{proof} \rangle$

lemma *weakest-intuitionistic*:

$\neg (\exists Q. (\forall h. ((\text{sep-true} \longrightarrow^* P) h \longrightarrow Q h)) \wedge \text{intuitionistic } Q \wedge$
 $Q \neq (\text{sep-true} \longrightarrow^* P) \wedge (\forall h. Q h \longrightarrow P h))$
 $\langle \text{proof} \rangle$

lemma *intuitionistic-sep-conj-sep-true-P*:

$\llbracket (P \wedge^* \text{sep-true}) s; \text{intuitionistic } P \rrbracket \Longrightarrow P s$
 $\langle \text{proof} \rangle$

lemma *intuitionistic-sep-conj-sep-true-simp*:

$\text{intuitionistic } P \Longrightarrow (P \wedge^* \text{sep-true}) = P$

<proof>

lemma *intuitionistic-sep-impl-sep-true-P*:
[[P h ; intuitionistic P]] \impl (*sep-true* $\longrightarrow^* P$) h
<proof>

lemma *intuitionistic-sep-impl-sep-true-simp*:
intuitionistic $P \impl$ (*sep-true* $\longrightarrow^* P$) = P
<proof>

4.12 Strictly exact assertions

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

lemma *strictly-exactD*:
[[*strictly-exact* P ; P h ; P h']] \impl $h = h'$
<proof>

lemma *strictly-exactI*:
($\bigwedge h h'. [[P h; P h']] \impl h = h'$) \impl *strictly-exact* P
<proof>

lemma *strictly-exact-sep-conj*:
[[*strictly-exact* P ; *strictly-exact* Q]] \impl *strictly-exact* ($P \wedge^* Q$)
<proof>

lemma *strictly-exact-conj-impl*:
[[($Q \wedge^* \text{sep-true}$) h ; P h ; *strictly-exact* Q]] \impl ($Q \wedge^* (Q \longrightarrow^* P)$) h
<proof>

end

interpretation *sep*: *ab-semigroup-mult* (**)
<proof>

interpretation *sep*: *comm-monoid-add* (**)
<proof>

5 Separation Algebra with Stronger, but More Intuitive Disjunction Axiom

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

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

```

    <proof>
subclass sep-algebra <proof>
end

```

6 Folding separating conjunction over lists of predicates

```

lemma sep-list-conj-Nil [simp]:  $\bigwedge^* [] = \square$ 
    <proof>

```

```

lemma (in semigroup-add) foldl-assoc:
shows foldl (+) (x+y) zs = x + (foldl (+) y zs)
    <proof>

```

```

lemma (in monoid-add) foldl-absorb0:
shows x + (foldl (+) 0 zs) = foldl (+) x zs
    <proof>

```

```

lemma sep-list-conj-Cons [simp]:  $\bigwedge^* (x\#xs) = (x ** \bigwedge^* xs)$ 
    <proof>

```

```

lemma sep-list-conj-append [simp]:  $\bigwedge^* (xs @ ys) = (\bigwedge^* xs ** \bigwedge^* ys)$ 
    <proof>

```

```

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

```

7 Separation Algebra with a Cancellative Monoid (for completeness)

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

```

class cancellative-sep-algebra = sep-algebra +
  assumes sep-add-cancelD:  $\llbracket x + z = y + z ; x \#\# z ; y \#\# z \rrbracket \implies x = y$ 
begin

```

```

definition

```

precise :: ('a ⇒ bool) ⇒ bool **where**
precise P = (∀ h hp hp'. hp ⚭ h ∧ P hp ∧ hp' ⚭ h ∧ P hp' ⟶ hp = hp')

lemma *precise* ((=) s)
 ⟨proof⟩

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

lemma *precise-distribute*:
precise P = (∀ Q R. ((Q and R) ∧* P) = ((Q ∧* P) and (R ∧* P)))
 ⟨proof⟩

lemma *strictly-precise*: *strictly-exact* P ⟹ *precise* P
 ⟨proof⟩

end

end

8 Standard Heaps as an Instance of Separation Algebra

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

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

class *opt* =
fixes *none* :: 'a
begin
definition *domain* f ≡ {x. f x ≠ *none*}
end

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

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

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

definition

plus-fun-def: $m1 + m2 \equiv \lambda x. \text{if } m2\ x = \text{none then } m1\ x \text{ else } m2\ x$

definition

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

instance

<proof>

end

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

lemma *domain-conv*: $\text{domain} = \text{dom}$

<proof>

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

<proof>

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

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

lemma

fixes *h* :: (*nat* => *nat*) => 'foo *option*

shows (*P* ** *Q* ** *H*) *h* = (*Q* ** *H* ** *P*) *h*

<proof>

end

9 Separation Logic Tactics

theory *Sep-Tactics*

imports *Separation-Algebra*

begin

<ML>

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

10 Selection (move-to-front) tactics

<ML>

11 Substitution

<ML>

12 Forward Reasoning

<ML>

13 Backward Reasoning

<ML>

14 Cancellation of Common Conjuncts via Elimination Rules

named-theorems *sep-cancel*

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

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

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

<ML>

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

<ML>

end

15 Example from HOL/Hoare/Separation

theory *Simple-Separation-Example*

imports *HOL-Hoare.Hoare-Logic-Abort* *../Sep-Heap-Instance*
../Sep-Tactics

begin

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

type-synonym $heap = (nat \Rightarrow nat\ option)$

definition $maps\ to :: nat \Rightarrow nat \Rightarrow heap \Rightarrow bool$ ($\langle - \mapsto - \rangle$ [56,51] 56)
where $x \mapsto y \equiv \lambda h. h = [x \mapsto y]$

notation $pred\ ex$ (**binder** $\langle \exists \rangle$ 10)

definition $maps\ to\ ex :: nat \Rightarrow heap \Rightarrow bool$ ($\langle - \mapsto - \rangle$ [56] 56)
where $x \mapsto - \equiv \exists y. x \mapsto y$

lemma $maps\ to\ maps\ to\ ex$ [elim]:
 $(p \mapsto v) s \Longrightarrow (p \mapsto -) s$
 $\langle proof \rangle$

lemma $maps\ to\ write$:
 $(p \mapsto - ** P) H \Longrightarrow (p \mapsto v ** P) (H (p \mapsto v))$
 $\langle proof \rangle$

lemma $points\ to$:
 $(p \mapsto v ** P) H \Longrightarrow the (H p) = v$
 $\langle proof \rangle$

primrec

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

lemma $list\ empty$ [simp]:
shows $list\ 0\ xs = (\lambda s. xs = [] \wedge \square s)$
 $\langle proof \rangle$

lemma $VARs$ $x\ y\ z\ w\ h$
 $\{(x \mapsto y ** z \mapsto w) h\}$
SKIP
 $\{x \neq z\}$
 $\langle proof \rangle$

lemma $VARs$ $H\ x\ y\ z\ w$

```

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

```

```

lemma VARS H
{p≠0 ∧ (p ↦ - ** list q qs) H}
H := H(p ↦ q)
{list p (p#qs) H}
⟨proof⟩

```

```

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}
⟨proof⟩

```

end

```

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

```

Substitution and forward/backward reasoning

```

typedecl p
typedecl val
typedecl heap

```

```

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

```

```

axiomatization
points-to :: p ⇒ 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)
⟨proof⟩

```

```

lemma
[[ Q2 (val h p); (K ** T ** blub ** P ** points-to p v ** P ** J) h ]]
⇒ Q (val h p) (val h p)

```

$\langle proof \rangle$

lemma

$\llbracket Q2 (val\ h\ p); (K ** T ** blub ** P ** points-to\ p\ v ** P ** J)\ h \rrbracket$
 $\implies Q (val\ h\ p) (val\ h\ p)$
 $\langle proof \rangle$

consts

$update :: p \Rightarrow val \Rightarrow heap \Rightarrow heap$

schematic-goal

assumes $a: \bigwedge P. (stuff\ p ** P)\ H \implies (other-stuff\ p\ v ** P)\ (update\ p\ v\ H)$
shows $(X ** Y ** other-stuff\ p\ ?v)\ (update\ p\ v\ H)$
 $\langle proof \rangle$

Example of low-level rewrites

lemma $\llbracket unrelated\ s ; (P ** Q ** R)\ s \rrbracket \implies (A ** B ** Q ** P)\ s$
 $\langle proof \rangle$

Conjunct selection

lemma $(A ** B ** Q ** P)\ s$
 $\langle proof \rangle$

lemma $\llbracket also\ unrelated; (A ** B ** Q ** P)\ s \rrbracket \implies unrelated$
 $\langle proof \rangle$

16 Test cases for *sep-cancel*.

lemma

assumes forward: $\bigwedge s\ g\ p\ v. A\ g\ p\ v\ s \implies AA\ g\ p\ s$
shows $\bigwedge xv\ yv\ P\ s\ y\ x\ s. (A\ g\ x\ yv ** A\ g\ y\ yv ** P)\ s \implies (AA\ g\ y ** sep-true)\ s$
 $\langle proof \rangle$

lemma

assumes forward: $\bigwedge s. generic\ s \implies instance\ s$
shows $(A ** generic ** B)\ s \implies (instance ** sep-true)\ s$
 $\langle proof \rangle$

lemma $\llbracket (A ** B)\ sa ; (A ** Y)\ s \rrbracket \implies (A ** X)\ s$
 $\langle proof \rangle$

lemma $\llbracket (A ** B)\ sa ; (A ** Y)\ s \rrbracket \implies (\lambda s. (A ** X)\ s)\ s$
 $\langle proof \rangle$

schematic-goal $\llbracket (B ** A ** C)\ s \rrbracket \implies (\lambda s. (A ** ?X)\ s)\ s$
 $\langle proof \rangle$

lemma
assumes *forward*: $\bigwedge s. \text{generic } s \implies \text{instance } s$
shows $\llbracket (A ** B) s ; (\text{generic} ** Y) s \rrbracket \implies (X ** \text{instance}) s$
 $\langle \text{proof} \rangle$

lemma
assumes *forward*: $\bigwedge s. \text{generic } s \implies \text{instance } s$
shows $\text{generic } s \implies \text{instance } s$
 $\langle \text{proof} \rangle$

lemma
assumes *forward*: $\bigwedge s. \text{generic } s \implies \text{instance } s$
assumes *forward2*: $\bigwedge s. \text{instance } s \implies \text{instance2 } s$
shows $\text{generic } s \implies (\text{instance2} ** \text{sep-true}) s$
 $\langle \text{proof} \rangle$

end

17 More properties of maps plus map disjunction.

theory *Map-Extra*
imports *Main*
begin

A note on naming: Anything not involving heap disjunction 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*: $(\text{None} \neq x) = (\exists y. x = \text{Some } y) \langle \text{proof} \rangle$

lemma *None-com*: $(\text{None} = x) = (x = \text{None}) \langle \text{proof} \rangle$

lemma *Some-com*: $(\text{Some } y = x) = (x = \text{Some } y) \langle \text{proof} \rangle$

19 Things that go into Map.thy

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

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

Map restriction via domain subtraction

definition

sub-restrict-map :: ('a → 'b) => 'a set => ('a → 'b) (**infixl** <'-> 110)

where

$m \text{ ' - } S \equiv (\lambda x. \text{ if } x \in S \text{ then None else } m \ x)$

19.1 Properties of maps not related to restriction

lemma *empty-forall-equiv*: $(m = \text{Map.empty}) = (\forall x. m \ x = \text{None})$
 <proof>

lemma *map-le-empty2* [*simp*]:
 $(m \subseteq_m \text{Map.empty}) = (m = \text{Map.empty})$
 <proof>

lemma *dom-iff*:
 $(\exists y. m \ x = \text{Some } y) = (x \in \text{dom } m)$
 <proof>

lemma *non-dom-eval*:
 $x \notin \text{dom } m \implies m \ x = \text{None}$
 <proof>

lemma *non-dom-eval-eq*:
 $x \notin \text{dom } m = (m \ x = \text{None})$
 <proof>

lemma *map-add-same-left-eq*:
 $m_1 = m_1' \implies (m_0 ++ m_1 = m_0 ++ m_1')$
 <proof>

lemma *map-add-left-cancelI* [*intro!*]:
 $m_1 = m_1' \implies m_0 ++ m_1 = m_0 ++ m_1'$
 <proof>

lemma *dom-empty-is-empty*:
 $(\text{dom } m = \{\}) = (m = \text{Map.empty})$
 <proof>

lemma *map-add-dom-eq*:
 $\text{dom } m = \text{dom } m' \implies m ++ m' = m'$
 <proof>

lemma *map-add-right-dom-eq*:
 $\llbracket m_0 ++ m_1 = m_0' ++ m_1'; \text{dom } m_1 = \text{dom } m_1' \rrbracket \implies m_1 = m_1'$
 <proof>

lemma *map-le-same-dom-eq*:
 $\llbracket m_0 \subseteq_m m_1; \text{dom } m_0 = \text{dom } m_1 \rrbracket \implies m_0 = m_1$
 <proof>

19.2 Properties of map restriction

lemma *restrict-map-cancel*:

$$(m \mid' S = m \mid' T) = (dom\ m \cap S = dom\ m \cap T)$$

<proof>

lemma *map-add-restricted-self* [*simp*]:

$$m ++ m \mid' S = m$$

<proof>

lemma *map-add-restrict-dom-right* [*simp*]:

$$(m ++ m') \mid' dom\ m' = m'$$

<proof>

lemma *restrict-map-UNIV* [*simp*]:

$$m \mid' UNIV = m$$

<proof>

lemma *restrict-map-dom*:

$$S = dom\ m \implies m \mid' S = m$$

<proof>

lemma *restrict-map-subdom*:

$$dom\ m \subseteq S \implies m \mid' S = m$$

<proof>

lemma *map-add-restrict*:

$$(m_0 ++ m_1) \mid' S = ((m_0 \mid' S) ++ (m_1 \mid' S))$$

<proof>

lemma *map-le-restrict*:

$$m \subseteq_m m' \implies m = m' \mid' dom\ m$$

<proof>

lemma *restrict-map-le*:

$$m \mid' S \subseteq_m m$$

<proof>

lemma *restrict-map-remerge*:

$$\llbracket S \cap T = \{\} \rrbracket \implies m \mid' S ++ m \mid' T = m \mid' (S \cup T)$$

<proof>

lemma *restrict-map-empty*:

$$dom\ m \cap S = \{\} \implies m \mid' S = Map.empty$$

<proof>

lemma *map-add-restrict-comp-right* [*simp*]:

$$(m \mid' S ++ m \mid' (UNIV - S)) = m$$

<proof>

lemma *map-add-restrict-comp-right-dom* [simp]:

$$(m \upharpoonright' S ++ m \upharpoonright' (\text{dom } m - S)) = m$$

<proof>

lemma *map-add-restrict-comp-left* [simp]:

$$(m \upharpoonright' (\text{UNIV} - S) ++ m \upharpoonright' S) = m$$

<proof>

lemma *restrict-self-UNIV*:

$$m \upharpoonright' (\text{dom } m - S) = m \upharpoonright' (\text{UNIV} - S)$$

<proof>

lemma *map-add-restrict-nonmember-right*:

$$x \notin \text{dom } m' \implies (m ++ m') \upharpoonright' \{x\} = m \upharpoonright' \{x\}$$

<proof>

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

$$x \notin \text{dom } m \implies (m ++ m') \upharpoonright' \{x\} = m' \upharpoonright' \{x\}$$

<proof>

lemma *map-add-restrict-right*:

$$x \subseteq \text{dom } m' \implies (m ++ m') \upharpoonright' x = m' \upharpoonright' x$$

<proof>

lemma *restrict-map-compose*:

$$\llbracket S \cup T = \text{dom } m ; S \cap T = \{\} \rrbracket \implies m \upharpoonright' S ++ m \upharpoonright' T = m$$

<proof>

lemma *map-le-dom-subset-restrict*:

$$\llbracket m' \subseteq_m m ; \text{dom } m' \subseteq S \rrbracket \implies m' \subseteq_m (m \upharpoonright' S)$$

<proof>

lemma *map-le-dom-restrict-sub-add*:

$$m' \subseteq_m m \implies m \upharpoonright' (\text{dom } m - \text{dom } m') ++ m' = m$$

<proof>

lemma *subset-map-restrict-sub-add*:

$$T \subseteq S \implies m \upharpoonright' (S - T) ++ m \upharpoonright' T = m \upharpoonright' S$$

<proof>

lemma *restrict-map-sub-union*:

$$m \upharpoonright' (\text{dom } m - (S \cup T)) = (m \upharpoonright' (\text{dom } m - T)) \upharpoonright' (\text{dom } m - S)$$

<proof>

lemma *prod-restrict-map-add*:

$$\llbracket S \cup T = U ; S \cap T = \{\} \rrbracket \implies m \upharpoonright' (X \times S) ++ m \upharpoonright' (X \times T) = m \upharpoonright' (X \times U)$$

<proof>

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

20.1 Definitions

Map disjunction

definition

$map\text{-}disj :: ('a \rightarrow 'b) \Rightarrow ('a \rightarrow 'b) \Rightarrow bool$ (**infix** $\langle \perp \rangle$ 51) **where**
 $h_0 \perp h_1 \equiv dom\ h_0 \cap dom\ h_1 = \{\}$

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

20.2 Properties of (\perp)

lemma *restrict-map-sub-disj*: $h \mid 'S \perp h \text{'- } S$
<proof>

lemma *restrict-map-sub-add*: $h \mid 'S ++ h \text{'- } S = h$
<proof>

20.3 Properties of map disjunction

lemma *map-disj-empty-right* [*simp*]:

$h \perp Map.empty$
<proof>

lemma *map-disj-empty-left* [*simp*]:

$Map.empty \perp h$
<proof>

lemma *map-disj-com*:

$h_0 \perp h_1 = h_1 \perp h_0$
<proof>

lemma *map-disjD*:

$h_0 \perp h_1 \implies dom\ h_0 \cap dom\ h_1 = \{\}$
<proof>

lemma *map-disjI*:

$dom\ h_0 \cap dom\ h_1 = \{\} \implies h_0 \perp h_1$
<proof>

20.4 Map associativity-commutativity based on map disjunction

lemma *map-add-com*:

$h_0 \perp h_1 \implies h_0 ++ h_1 = h_1 ++ h_0$
<proof>

lemma *map-add-left-commute*:

$$h_0 \perp h_1 \implies h_0 ++ (h_1 ++ h_2) = h_1 ++ (h_0 ++ h_2)$$

<proof>

lemma *map-add-disj*:

$$h_0 \perp (h_1 ++ h_2) = (h_0 \perp h_1 \wedge h_0 \perp h_2)$$

<proof>

lemma *map-add-disj'*:

$$(h_1 ++ h_2) \perp h_0 = (h_1 \perp h_0 \wedge h_2 \perp h_0)$$

<proof>

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

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

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

lemmas *map-add-ac* =

$$\text{map-add-assoc[symmetric] map-add-com map-disj-com map-add-left-commute map-add-disj map-add-disj'}$$

20.5 Basic properties

lemma *map-disj-None-right*:

$$\llbracket h_0 \perp h_1 ; x \in \text{dom } h_0 \rrbracket \implies h_1 x = \text{None}$$

<proof>

lemma *map-disj-None-left*:

$$\llbracket h_0 \perp h_1 ; x \in \text{dom } h_1 \rrbracket \implies h_0 x = \text{None}$$

<proof>

lemma *map-disj-None-left'*:

$$\llbracket h_0 x = \text{Some } y ; h_1 \perp h_0 \rrbracket \implies h_1 x = \text{None}$$

<proof>

lemma *map-disj-None-right'*:

$$\llbracket h_1 x = \text{Some } y ; h_1 \perp h_0 \rrbracket \implies h_0 x = \text{None}$$

<proof>

lemma *map-disj-common*:

$$\llbracket h_0 \perp h_1 ; h_0 p = \text{Some } v ; h_1 p = \text{Some } v' \rrbracket \implies \text{False}$$

<proof>

lemma *map-disj-eq-dom-left*:

$$\llbracket h_0 \perp h_1 ; \text{dom } h_0' = \text{dom } h_0 \rrbracket \implies h_0' \perp h_1$$

<proof>

20.6 Map disjunction and addition

lemma *map-add-eval-left*:

$\llbracket x \in \text{dom } h ; h \perp h' \rrbracket \implies (h ++ h') x = h x$
 $\langle \text{proof} \rangle$

lemma *map-add-eval-right*:

$\llbracket x \in \text{dom } h' ; h \perp h' \rrbracket \implies (h ++ h') x = h' x$
 $\langle \text{proof} \rangle$

lemma *map-add-eval-left'*:

$\llbracket x \notin \text{dom } h' ; h \perp h' \rrbracket \implies (h ++ h') x = h x$
 $\langle \text{proof} \rangle$

lemma *map-add-eval-right'*:

$\llbracket x \notin \text{dom } h ; h \perp h' \rrbracket \implies (h ++ h') x = h' x$
 $\langle \text{proof} \rangle$

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

assumes *eq*: $h_0 ++ h_1 = h_0' ++ h_1'$
assumes *etc*: $h_0 \perp h_1 \ h_0' \perp h_1' \ \text{dom } h_0 = \text{dom } h_0'$
shows $h_0 = h_0'$
 $\langle \text{proof} \rangle$

lemma *map-add-left-eq*:

assumes *eq*: $h_0 ++ h = h_1 ++ h$
assumes *disj*: $h_0 \perp h \ h_1 \perp h$
shows $h_0 = h_1$
 $\langle \text{proof} \rangle$

lemma *map-add-right-eq*:

$\llbracket h ++ h_0 = h ++ h_1 ; h_0 \perp h ; h_1 \perp h \rrbracket \implies h_0 = h_1$
 $\langle \text{proof} \rangle$

lemma *map-disj-add-eq-dom-right-eq*:

assumes *merge*: $h_0 ++ h_1 = h_0' ++ h_1'$ **and** *d*: $\text{dom } h_0 = \text{dom } h_0'$ **and**
ab-disj: $h_0 \perp h_1$ **and** *cd-disj*: $h_0' \perp h_1'$
shows $h_1 = h_1'$
 $\langle \text{proof} \rangle$

lemma *map-disj-add-eq-dom-left-eq*:

assumes *add*: $h_0 ++ h_1 = h_0' ++ h_1'$ **and**
dom: $\text{dom } h_1 = \text{dom } h_1'$ **and**
disj: $h_0 \perp h_1 \ h_0' \perp h_1'$
shows $h_0 = h_0'$
 $\langle \text{proof} \rangle$

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>

lemma *map-add-lr-disj*:

$$\llbracket h_0 ++ h_1 = h_0' ++ h_1'; h_1 \perp h_1' \rrbracket \implies \text{dom } h_1 \subseteq \text{dom } h_0'$$

<proof>

20.7 Map disjunction and map updates

lemma *map-disj-update-left* [*simp*]:

$$p \in \text{dom } h_1 \implies h_0 \perp h_1(p \mapsto v) = h_0 \perp h_1$$

<proof>

lemma *map-disj-update-right* [*simp*]:

$$p \in \text{dom } h_1 \implies h_1(p \mapsto v) \perp h_0 = h_1 \perp h_0$$

<proof>

lemma *map-add-update-left*:

$$\llbracket h_0 \perp h_1; p \in \text{dom } h_0 \rrbracket \implies (h_0 ++ h_1)(p \mapsto v) = (h_0(p \mapsto v) ++ h_1)$$

<proof>

lemma *map-add-update-right*:

$$\llbracket h_0 \perp h_1; p \in \text{dom } h_1 \rrbracket \implies (h_0 ++ h_1)(p \mapsto v) = (h_0 ++ h_1(p \mapsto v))$$

<proof>

lemma *map-add3-update*:

$$\llbracket h_0 \perp h_1; h_1 \perp h_2; h_0 \perp h_2; p \in \text{dom } h_0 \rrbracket \\ \implies (h_0 ++ h_1 ++ h_2)(p \mapsto v) = h_0(p \mapsto v) ++ h_1 ++ h_2$$

<proof>

20.8 Map disjunction and (\subseteq_m)

lemma *map-le-override* [*simp*]:

$$\llbracket h \perp h' \rrbracket \implies h \subseteq_m h ++ h'$$

<proof>

lemma *map-leI-left*:

$$\llbracket h = h_0 ++ h_1; h_0 \perp h_1 \rrbracket \implies h_0 \subseteq_m h \text{ } \langle \text{proof} \rangle$$

lemma *map-leI-right*:

$$\llbracket h = h_0 ++ h_1; h_0 \perp h_1 \rrbracket \implies h_1 \subseteq_m h \text{ } \langle \text{proof} \rangle$$

lemma *map-disj-map-le*:

$$\llbracket h_0' \subseteq_m h_0; h_0 \perp h_1 \rrbracket \implies h_0' \perp h_1$$

<proof>

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

$$\llbracket h' \subseteq_m h; h_0 \perp h_1; h' = h_0 ++ h_1 \rrbracket \implies h_0 \subseteq_m h$$

<proof>

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

$\llbracket h' \subseteq_m h ; h_0 \perp h_1 ; h' = h_1 ++ h_0 \rrbracket \implies h_0 \subseteq_m h$
 $\langle proof \rangle$

lemma *map-le-add-cancel*:

$\llbracket h_0 \perp h_1 ; h_0' \subseteq_m h_0 \rrbracket \implies h_0' ++ h_1 \subseteq_m h_0 ++ h_1$
 $\langle proof \rangle$

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 $h_0' \subseteq_m h_0$

$\langle proof \rangle$

lemma *map-le-conv*:

$(h_0' \subseteq_m h_0 \wedge h_0' \neq h_0) = (\exists h_1. h_0 = h_0' ++ h_1 \wedge h_0' \perp h_1 \wedge h_0' \neq h_0)$

$\langle proof \rangle$

lemma *map-le-conv2*:

$h_0' \subseteq_m h_0 = (\exists h_1. h_0 = h_0' ++ h_1 \wedge h_0' \perp h_1)$

$\langle proof \rangle$

20.9 Map disjunction and restriction

lemma *map-disj-comp* [*simp*]:

$h_0 \perp h_1 \mid' (UNIV - \text{dom } h_0)$

$\langle proof \rangle$

lemma *restrict-map-disj*:

$S \cap T = \{\} \implies h \mid' S \perp h \mid' T$

$\langle proof \rangle$

lemma *map-disj-restrict-dom* [*simp*]:

$h_0 \perp h_1 \mid' (\text{dom } h_1 - \text{dom } h_0)$

$\langle proof \rangle$

lemma *restrict-map-disj-dom-empty*:

$h \perp h' \implies h \mid' \text{dom } h' = \text{Map.empty}$

$\langle proof \rangle$

lemma *restrict-map-univ-disj-eq*:

$h \perp h' \implies h \mid' (UNIV - \text{dom } h') = h$

$\langle proof \rangle$

lemma *restrict-map-disj-dom*:

$h_0 \perp h_1 \implies h \mid' \text{dom } h_0 \perp h \mid' \text{dom } h_1$

$\langle proof \rangle$

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

$h \perp h' \implies (h ++ h') \mid' \text{dom } h = h$
 ⟨proof⟩

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

$h \perp h' \implies S = \text{dom } h \implies (h ++ h') \mid' S = h$
 ⟨proof⟩

lemma *restrict-map-disj-left*:

$h_0 \perp h_1 \implies h_0 \mid' S \perp h_1$
 ⟨proof⟩

lemma *restrict-map-disj-right*:

$h_0 \perp h_1 \implies h_0 \perp h_1 \mid' S$
 ⟨proof⟩

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

lemma *map-dom-disj-restrict-right*:

$h_0 \perp h_1 \implies (h_0 ++ h_0') \mid' \text{dom } h_1 = h_0' \mid' \text{dom } h_1$
 ⟨proof⟩

lemma *restrict-map-on-disj*:

$h_0' \perp h_1 \implies h_0 \mid' \text{dom } h_0' \perp h_1$
 ⟨proof⟩

lemma *restrict-map-on-disj'*:

$h_0 \perp h_1 \implies h_0 \perp h_1 \mid' S$
 ⟨proof⟩

lemma *map-le-sub-dom*:

$\llbracket h_0 ++ h_1 \subseteq_m h ; h_0 \perp h_1 \rrbracket \implies h_0 \subseteq_m h \mid' (\text{dom } h - \text{dom } h_1)$
 ⟨proof⟩

lemma *map-submap-break*:

$\llbracket h \subseteq_m h' \rrbracket \implies h' = (h' \mid' (\text{UNIV} - \text{dom } h)) ++ h$
 ⟨proof⟩

lemma *map-add-disj-restrict-both*:

$\llbracket h_0 \perp h_1 ; S \cap S' = \{\}; T \cap T' = \{\} \rrbracket$
 $\implies (h_0 \mid' S) ++ (h_1 \mid' T) \perp (h_0 \mid' S') ++ (h_1 \mid' T')$
 ⟨proof⟩

end

21 Separation Algebra for Virtual Memory

theory *VM-Example*

imports *../Sep-Tactics ../Map-Extra*

begin

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

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

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

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

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

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

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

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

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

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

```
instance
  ⟨proof⟩
```

```
end
```

end

22 Abstract Separation Logic, Alternative Definition

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

This theory contains an alternative definition of separation 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** $\langle ++ \rangle$ 100)

definition

lift2 :: ('a => 'b => 'c option) => 'a option => 'b option => 'c option

where

lift2 f a b \equiv case (a,b) of (Some a, Some b) => f a b | - => None

class *sep-algebra-alt* = zero +
fixes *add* :: 'a => 'a => 'a option (**infixr** $\langle \oplus \rangle$ 65)

assumes *add-zero* [*simp*]: $x \oplus 0 = \text{Some } x$

assumes *add-comm*: $x \oplus y = y \oplus x$

assumes *add-assoc*: *lift2* add a (*lift2* add b c) = *lift2* add (*lift2* add a b) c

begin

definition

disjoint :: 'a => 'a => bool (**infix** $\langle \#\# \rangle$ 60)

where

$a \#\# b \equiv a \oplus b \neq \text{None}$

lemma *disj-com*: $x \#\# y = y \#\# x$

<proof>

lemma *disj-zero* [*simp*]: $x \#\# 0$

<proof>

lemma *disj-zero2* [*simp*]: $0 \#\# x$

<proof>

lemma *add-zero2* [*simp*]: $0 \oplus x = \text{Some } x$

<proof>

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* ≡ λs. ∃ p q. p ⊕ q = Some s ∧ P p ∧ Q q

definition *emp* :: 'a ⇒ bool (<□>) **where**

□ ≡ λs. s = 0

definition

sep-impl :: ('a ⇒ bool) ⇒ ('a ⇒ bool) ⇒ ('a ⇒ bool) (**infixr** <⟶*⟩ 25)
where
P ⟶ Q* ≡ λh. ∀ h' h''. h ⊕ h' = Some h'' ∧ P h' ⟶ Q h''

definition (**in** -)

sep-true ≡ λs. True

definition (**in** -)

sep-false ≡ λs. False

abbreviation

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

lemma *add2-comm*:

a ++ b = b ++ a
 <proof>

lemma *add2-None* [*simp*]:

x ++ None = None
 <proof>

lemma *None-add2* [*simp*]:

None ++ x = None
 <proof>

lemma *add2-Some-Some*:

Some x ++ Some y = x ⊕ y
 <proof>

lemma *add2-zero* [*simp*]:

Some x ++ Some 0 = Some x
 <proof>

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

lemma *sep-conjE*:
 $\llbracket (P ** Q) s; \bigwedge p q. \llbracket P p; Q q; p \oplus q = \text{Some } s \rrbracket \implies X \rrbracket \implies X$
 ⟨*proof*⟩

lemma *sep-conjI*:
 $\llbracket P p; Q q; p \oplus q = \text{Some } s \rrbracket \implies (P ** Q) s$
 ⟨*proof*⟩

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

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

lemma *lift-to-add2*:
 $\llbracket z \oplus q = \text{Some } s; x \oplus y = \text{Some } q \rrbracket \implies \text{Some } z ++ \text{Some } x ++ \text{Some } y = \text{Some } s$
 ⟨*proof*⟩

lemma *lift-to-add2'*:
 $\llbracket q \oplus z = \text{Some } s; x \oplus y = \text{Some } q \rrbracket \implies (\text{Some } x ++ \text{Some } y) ++ \text{Some } z = \text{Some } s$
 ⟨*proof*⟩

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

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

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

lemma (**in** $-$) *sep-true*[*simp*]: *sep-true s* ⟨*proof*⟩

lemma (**in** $-$) *sep-false*[*simp*]: $\neg \text{sep-false } x$ ⟨*proof*⟩

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

lemma *sep-conj-sep-true'*:

$$P s \implies (\text{sep-true} ** P) s$$

<proof>

lemma *disjoint-submaps-exist*:

$$\exists h_0 h_1. h_0 \oplus h_1 = \text{Some } h$$

<proof>

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

$$(\text{sep-true} ** \text{sep-true}) = \text{sep-true}$$

<proof>

lemma *sep-conj-false-right[simp]*:

$$(P ** \text{sep-false}) = \text{sep-false}$$

<proof>

lemma *sep-conj-false-left[simp]*:

$$(\text{sep-false} ** P) = \text{sep-false}$$

<proof>

lemma *sep-conj-left-com*:

$$(P ** (Q ** R)) = (Q ** (P ** R)) \text{ (is } ?x = ?y)$$

<proof>

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

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

<proof>

lemma *sep-conj-empty[simp]*:

$$(P ** \square) = P$$

<proof>

lemma *sep-conj-empty'[simp]*:

$$(\square ** P) = P$$

<proof>

lemma *sep-conj-sep-emptyI*:

$$P s \implies (P ** \square) s$$

<proof>

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

$$(\text{sep-true} ** (\text{sep-true} ** P)) = (\text{sep-true} ** P)$$

<proof>

lemma *sep-conj-disj*:

$$((\lambda s. P s \vee Q s) ** R) s = ((P ** R) s \vee (Q ** R) s) \text{ (is } ?x = (?y \vee ?z))$$

<proof>

lemma *sep-conj-conj*:

$$((\lambda s. P s \wedge Q s) ** R) s \implies (P ** R) s \wedge (Q ** R) s$$

<proof>

lemma *sep-conj-exists1*:

$$((\lambda s. \exists x. P x s) ** Q) s = (\exists x. (P x ** Q) s)$$

<proof>

lemma *sep-conj-exists2*:

$$(P ** (\lambda s. \exists x. Q x s)) = (\lambda s. (\exists x. (P ** Q x) s))$$

<proof>

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

lemma *sep-conj-forall*:

$$((\lambda s. \forall x. P x s) ** Q) s \implies (P x ** Q) s$$

<proof>

lemma *sep-conj-impl*:

$$\llbracket (P ** Q) s; \bigwedge s. P s \implies P' s; \bigwedge s. Q s \implies Q' s \rrbracket \implies (P' ** Q') s$$

<proof>

lemma *sep-conj-impl1*:

assumes $P: \bigwedge s. P s \implies I s$
shows $(P ** R) s \implies (I ** R) s$
<proof>

lemma *sep-conj-sep-true-left*:

$$(P ** Q) s \implies (sep-true ** Q) s$$

<proof>

lemma *sep-conj-sep-true-right*:

$$(P ** Q) s \implies (P ** sep-true) s$$

<proof>

lemma *sep-globalise*:

$$\llbracket (P ** R) s; \bigwedge s. P s \implies Q s \rrbracket \implies (Q ** R) s$$

<proof>

lemma *sep-implI*:

assumes $a: \bigwedge h' h''. \llbracket h \oplus h' = \text{Some } h''; P h' \rrbracket \implies Q h''$
shows $(P \longrightarrow^* Q) h$
<proof>

lemma *sep-implD*:

$$(x \longrightarrow^* y) h \implies \forall h' h''. h \oplus h' = \text{Some } h'' \wedge x h' \longrightarrow y h''$$

<proof>

lemma *sep-impl-sep-true*[simp]:
 $(P \longrightarrow^* \text{sep-true}) = \text{sep-true}$
 $\langle \text{proof} \rangle$

lemma *sep-impl-sep-false*[simp]:
 $(\text{sep-false} \longrightarrow^* P) = \text{sep-true}$
 $\langle \text{proof} \rangle$

lemma *sep-impl-sep-true-P*:
 $(\text{sep-true} \longrightarrow^* P) s \implies P s$
 $\langle \text{proof} \rangle$

lemma *sep-impl-sep-true-false*[simp]:
 $(\text{sep-true} \longrightarrow^* \text{sep-false}) = \text{sep-false}$
 $\langle \text{proof} \rangle$

lemma *sep-conj-sep-impl*:
 $\llbracket P s; \bigwedge s. (P ** Q) s \implies R s \rrbracket \implies (Q \longrightarrow^* R) s$
 $\langle \text{proof} \rangle$

lemma *sep-conj-sep-impl2*:
 $\llbracket (P ** Q) s; \bigwedge s. P s \implies (Q \longrightarrow^* R) s \rrbracket \implies R s$
 $\langle \text{proof} \rangle$

lemma *sep-conj-sep-impl-sep-conj2*:
 $(P ** R) s \implies (P ** (Q \longrightarrow^* (Q ** R))) s$
 $\langle \text{proof} \rangle$

lemma *sep-conj-triv-strip2*:
 $Q = R \implies (Q ** P) = (R ** P) \langle \text{proof} \rangle$

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** $\langle ++ \rangle$ 100)

24 Total implies Partial

definition $add2 :: 'a::sep-algebra \Rightarrow 'a \Rightarrow 'a$ option **where**
 $add2\ x\ y \equiv$ if $x \#\# y$ then $Some\ (x + y)$ else $None$

lemma $add2-zero$: $add2\ x\ 0 = Some\ x$
(proof)

lemma $add2-comm$: $add2\ x\ y = add2\ y\ x$
(proof)

lemma $add2-assoc$:
 $lift2\ add2\ a\ (lift2\ add2\ b\ c) = lift2\ add2\ (lift2\ add2\ a\ b)\ c$
(proof)

interpretation $total-partial$: $sep-algebra-alt\ 0\ add2$
(proof)

25 Partial implies Total

definition
 $sep-add' :: 'a \Rightarrow 'a \Rightarrow 'a :: sep-algebra-alt$ **where**
 $sep-add'\ x\ y \equiv$ if disjoint $x\ y$ then the $(add\ x\ y)$ else undefined

lemma $sep-disj-zero'$:
 $disjoint\ x\ 0$
(proof)

lemma $sep-disj-commuteI'$:
 $disjoint\ x\ y \Longrightarrow disjoint\ y\ x$
(proof)

lemma $sep-add-zero'$:
 $sep-add'\ x\ 0 = x$
(proof)

lemma $sep-add-commute'$:
 $disjoint\ x\ y \Longrightarrow sep-add'\ x\ y = sep-add'\ y\ x$
(proof)

lemma $sep-add-assoc'$:
 $\llbracket disjoint\ x\ y; disjoint\ y\ z; disjoint\ x\ z \rrbracket \Longrightarrow$
 $sep-add'\ (sep-add'\ x\ y)\ z = sep-add'\ x\ (sep-add'\ y\ z)$
(proof)

lemma $sep-disj-addD1'$:
 $disjoint\ x\ (sep-add'\ y\ z) \Longrightarrow disjoint\ y\ z \Longrightarrow disjoint\ x\ y$
(proof)

```

lemma sep-disj-addI1':
  disjoint x (sep-add' y z)  $\implies$  disjoint y z  $\implies$  disjoint (sep-add' x y) z
  <proof>

interpretation partial-total: sep-algebra sep-add' 0 disjoint
  <proof>

end

```

26 A simplified version of the actual capDL specification.

```

theory Types-D
imports HOL-Library.Word
begin

type-synonym cdl-object-id = 32 word

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

type-synonym cdl-size-bits = nat

type-synonym cdl-cnode-index = nat

type-synonym cdl-cap-ref = cdl-object-id  $\times$  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  $\Rightarrow$  cdl-cap option

translations
  (type) cdl-cap-map <= (type) nat  $\Rightarrow$  cdl-cap option
  (type) cdl-cap-ref <= (type) cdl-object-id  $\times$  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* \Rightarrow *cdl-object option*

type-synonym *cdl-component* = *nat option*

type-synonym *cdl-components* = *cdl-component set*

type-synonym *cdl-ghost-state* = *cdl-object-id* \Rightarrow *cdl-components*

translations

(*type*) *cdl-heap* \leq (*type*) *cdl-object-id* \Rightarrow *cdl-object option*

(*type*) *cdl-ghost-state* \leq (*type*) *cdl-object-id* \Rightarrow *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* \Rightarrow *cdl-object-type*

where

object-type *x* \equiv
 case *x* *of*

$Endpoint \Rightarrow EndpointType$
 $| Tcb - \Rightarrow TcbType$
 $| CNode - \Rightarrow CNodeType$

definition $cap\text{-}objects :: cdl\text{-}cap \Rightarrow cdl\text{-}object\text{-}id\ set$

where

$cap\text{-}objects\ cap \equiv$
 $case\ cap\ of$
 $TcbCap\ x \Rightarrow \{x\}$
 $| CNodeCap\ x \Rightarrow \{x\}$
 $| EndpointCap\ x - \Rightarrow \{x\}$

definition $cap\text{-}has\text{-}object :: cdl\text{-}cap \Rightarrow bool$

where

$cap\text{-}has\text{-}object\ cap \equiv$
 $case\ cap\ of$
 $NullCap \quad \Rightarrow False$
 $| - \quad \quad \Rightarrow True$

definition $cap\text{-}object :: cdl\text{-}cap \Rightarrow cdl\text{-}object\text{-}id$

where

$cap\text{-}object\ cap \equiv$
 $if\ cap\text{-}has\text{-}object\ cap$
 $then\ THE\ obj\text{-}id.\ cap\text{-}objects\ cap = \{obj\text{-}id\}$
 $else\ undefined$

lemma $cap\text{-}object\text{-}simps:$

$cap\text{-}object\ (TcbCap\ x) = x$
 $cap\text{-}object\ (CNodeCap\ x) = x$
 $cap\text{-}object\ (EndpointCap\ x\ j) = x$
 $\langle proof \rangle$

definition

$cap\text{-}rights :: cdl\text{-}cap \Rightarrow cdl\text{-}right\ set$

where

$cap\text{-}rights\ c \equiv case\ c\ of$
 $EndpointCap\ -\ x \Rightarrow x$
 $| - \Rightarrow UNIV$

definition

$update\text{-}cap\text{-}rights :: cdl\text{-}right\ set \Rightarrow cdl\text{-}cap \Rightarrow cdl\text{-}cap$

where

$update\text{-}cap\text{-}rights\ r\ c \equiv case\ c\ of$
 $EndpointCap\ f1\ - \Rightarrow EndpointCap\ f1\ r$
 $| - \Rightarrow c$

definition

object-slots :: *cdl-object* \Rightarrow *cdl-cap-map*

where

object-slots obj \equiv *case obj of*
CNode x \Rightarrow *cdl-cnode-caps x*
 | *Tcb x* \Rightarrow *cdl-tcb-caps x*
 | - \Rightarrow *Map.empty*

definition

update-slots :: *cdl-cap-map* \Rightarrow *cdl-object* \Rightarrow *cdl-object*

where

update-slots new-val obj \equiv *case obj of*
CNode x \Rightarrow *CNode (x (cdl-cnode-caps := new-val))*
 | *Tcb x* \Rightarrow *Tcb (x (cdl-tcb-caps := new-val))*
 | - \Rightarrow *obj*

definition

add-to-slots :: *cdl-cap-map* \Rightarrow *cdl-object* \Rightarrow *cdl-object*

where

add-to-slots new-val obj \equiv *update-slots (new-val ++ (object-slots obj)) obj*

definition

slots-of :: *cdl-heap* \Rightarrow *cdl-object-id* \Rightarrow *cdl-cap-map*

where

slots-of h \equiv λ *obj-id.*
case h obj-id of
None \Rightarrow *Map.empty*
 | *Some obj* \Rightarrow *object-slots obj*

definition

has-slots :: *cdl-object* \Rightarrow *bool*

where

has-slots obj \equiv *case obj of*
CNode - \Rightarrow *True*
 | *Tcb -* \Rightarrow *True*
 | - \Rightarrow *False*

definition

object-at :: (*cdl-object* \Rightarrow *bool*) \Rightarrow *cdl-object-id* \Rightarrow *cdl-heap* \Rightarrow *bool*

where

object-at P p s \equiv \exists *object.* *s p = Some object* \wedge *P object*

abbreviation

ko-at k \equiv *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:
[[ $x \in A$ ;  $A \cap B = \{\}$ ]]  $\implies x \notin B$ 
<proof>
```

```
lemma union-intersection:
 $A \cap (A \cup B) = A$ 
 $B \cap (A \cup B) = B$ 
 $(A \cup B) \cap A = A$ 
 $(A \cup B) \cap B = B$ 
<proof>
```

```
lemma union-intersection1:  $A \cap (A \cup B) = A$ 
<proof>
```

```
lemma union-intersection2:  $B \cap (A \cup B) = B$ 
<proof>
```

```
lemma restrict-map-disj':
 $S \cap T = \{\} \implies h \mid' S \perp h' \mid' T$ 
<proof>
```

```
lemma map-add-restrict-comm:
 $S \cap T = \{\} \implies h \mid' S ++ h' \mid' T = h' \mid' T ++ h \mid' S$ 
<proof>
```

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

```
primrec sep-heap :: sep-state  $\Rightarrow$  cdl-heap
where sep-heap (SepState h gs) = h
```

```
primrec sep-ghost-state :: sep-state  $\Rightarrow$  cdl-ghost-state
where sep-ghost-state (SepState h gs) = gs
```

definition

$the\text{-}set :: 'a\ option\ set \Rightarrow 'a\ set$

where

$the\text{-}set\ xs = \{x. Some\ x \in xs\}$

lemma *the-set-union* [simp]:

$the\text{-}set\ (A \cup B) = the\text{-}set\ A \cup the\text{-}set\ B$
 ⟨proof⟩

lemma *the-set-inter* [simp]:

$the\text{-}set\ (A \cap B) = the\text{-}set\ A \cap the\text{-}set\ B$
 ⟨proof⟩

lemma *the-set-inter-empty*:

$A \cap B = \{\} \Longrightarrow the\text{-}set\ A \cap the\text{-}set\ B = \{\}$
 ⟨proof⟩

definition

$slots\text{-}of\text{-}heap :: cdl\text{-}heap \Rightarrow cdl\text{-}object\text{-}id \Rightarrow cdl\text{-}cap\text{-}map$

where

$slots\text{-}of\text{-}heap\ h \equiv \lambda obj\text{-}id.$

$case\ h\ obj\text{-}id\ of$

$None \Rightarrow Map.empty$

$| Some\ obj \Rightarrow object\text{-}slots\ obj$

definition

$add\text{-}to\text{-}slots :: cdl\text{-}cap\text{-}map \Rightarrow cdl\text{-}object \Rightarrow cdl\text{-}object$

where

$add\text{-}to\text{-}slots\ new\text{-}val\ obj \equiv update\text{-}slots\ (new\text{-}val\ ++\ (object\text{-}slots\ obj))\ obj$

lemma *add-to-slots-assoc*:

$add\text{-}to\text{-}slots\ x\ (add\text{-}to\text{-}slots\ (y\ ++\ z)\ obj) =$
 $add\text{-}to\text{-}slots\ (x\ ++\ y)\ (add\text{-}to\text{-}slots\ z\ obj)$
 ⟨proof⟩

lemma *add-to-slots-twice* [simp]:

$add\text{-}to\text{-}slots\ x\ (add\text{-}to\text{-}slots\ y\ a) = add\text{-}to\text{-}slots\ (x\ ++\ y)\ a$
 ⟨proof⟩

lemma *slots-of-heap-empty* [simp]: $slots\text{-}of\text{-}heap\ Map.empty\ object\text{-}id = Map.empty$

⟨proof⟩

lemma *slots-of-heap-empty2* [simp]:

$h\ obj\text{-}id = None \Longrightarrow slots\text{-}of\text{-}heap\ h\ obj\text{-}id = Map.empty$

<proof>

lemma *update-slots-add-to-slots-empty* [simp]:

update-slots Map.empty (add-to-slots new obj) = update-slots Map.empty obj

<proof>

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

<proof>

lemma *update-slots-of-heap-id* [simp]:

h obj-id = Some obj \implies update-slots (slots-of-heap h obj-id) obj = obj

<proof>

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

<proof>

lemma *update-slots-eq*:

update-slots a o1 = update-slots a o2 \implies update-slots b o1 = update-slots b o2

<proof>

definition

not-conflicting-objects :: sep-state \Rightarrow sep-state \Rightarrow cdl-object-id \Rightarrow 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) \Rightarrow object-type o1 = object-type o2 \wedge gs-a obj-id \cap gs-b

obj-id = {}

| - \Rightarrow True)

definition

clean-slots :: cdl-cap-map \Rightarrow cdl-components \Rightarrow cdl-cap-map

where

clean-slots slots cmp \equiv slots |' the-set cmp

definition

object-clean-fields :: cdl-object \Rightarrow cdl-components \Rightarrow cdl-object

where

object-clean-fields obj cmp \equiv if None \in cmp then obj else case obj of

Tcb x \Rightarrow 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* \Rightarrow *sep-state* \Rightarrow *cdl-ghost-state*
where
cdl-ghost-state-add state-a state-b \equiv 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 \wedge heap-b obj-id \neq None then gs-b obj-id
else if heap-b obj-id = None \wedge heap-a obj-id \neq None then gs-a obj-id
else gs-a obj-id \cup gs-b obj-id

definition

sep-state-add :: *sep-state* \Rightarrow *sep-state* \Rightarrow *sep-state*
where
sep-state-add state-a state-b \equiv
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* \Rightarrow *sep-state* \Rightarrow *bool*
where
sep-state-disj state-a state-b \equiv
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 \forall obj-id. not-conflicting-objects state-a state-b obj-id

lemma not-conflicting-objects-comm:

not-conflicting-objects h1 h2 obj = not-conflicting-objects h2 h1 obj
<proof>

lemma object-clean-comm:

\llbracket *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 \notin cmp
 \implies *object-clean (add-to-slots (object-slots obj-a) obj-b) cmp =*
*object-clean (add-to-slots (object-slots obj-b) obj-a) cmp**

<proof>

lemma *add-to-slots-object-slots*:

object-type y = object-type z

$\implies \text{add-to-slots } (\text{object-slots } (\text{add-to-slots } (x) y)) z =$

$\text{add-to-slots } (x ++ \text{object-slots } y) z$

<proof>

lemma *not-conflicting-objects-empty* [*simp*]:

not-conflicting-objects s (SepState Map.empty (λobj-id. {})) obj-id

<proof>

lemma *empty-not-conflicting-objects* [*simp*]:

not-conflicting-objects (SepState Map.empty (λobj-id. {})) s obj-id

<proof>

lemma *not-conflicting-objects-empty-object* [*elim!*]:

(sep-heap x) obj-id = None \implies not-conflicting-objects x y obj-id

<proof>

lemma *empty-object-not-conflicting-objects* [*elim!*]:

(sep-heap y) obj-id = None \implies not-conflicting-objects x y obj-id

<proof>

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

cdl-heap-add (SepState h gs) (SepState Map.empty (λobj-id. {})) = h

<proof>

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

cdl-heap-add (SepState Map.empty (λobj-id. {})) (SepState h gs) = h

<proof>

lemma *map-add-result-empty1*: $a ++ b = \text{Map.empty} \implies a = \text{Map.empty}$

<proof>

lemma *map-add-result-empty2*: $a ++ b = \text{Map.empty} \implies b = \text{Map.empty}$

<proof>

lemma *map-add-emptyE* [*elim!*]: $\llbracket a ++ b = \text{Map.empty}; \llbracket a = \text{Map.empty}; b = \text{Map.empty} \rrbracket \implies R \rrbracket \implies R$

<proof>

lemma *clean-slots-empty* [*simp*]:

clean-slots Map.empty cmp = Map.empty

<proof>

lemma *object-type-update-slots* [*simp*]:

object-type (update-slots slots x) = object-type x

<proof>

lemma *object-type-object-clean-slots* [simp]:
 $\text{object-type } (\text{object-clean-slots } x \text{ cmp}) = \text{object-type } x$
 ⟨proof⟩

lemma *object-type-object-clean-fields* [simp]:
 $\text{object-type } (\text{object-clean-fields } x \text{ cmp}) = \text{object-type } x$
 ⟨proof⟩

lemma *object-type-object-clean* [simp]:
 $\text{object-type } (\text{object-clean } x \text{ cmp}) = \text{object-type } x$
 ⟨proof⟩

lemma *object-type-add-to-slots* [simp]:
 $\text{object-type } (\text{add-to-slots } \text{slots } x) = \text{object-type } x$
 ⟨proof⟩

lemma *object-slots-update-slots* [simp]:
 $\text{has-slots } \text{obj} \implies \text{object-slots } (\text{update-slots } \text{slots } \text{obj}) = \text{slots}$
 ⟨proof⟩

lemma *object-slots-update-slots-empty* [simp]:
 $\neg \text{has-slots } \text{obj} \implies \text{object-slots } (\text{update-slots } \text{slots } \text{obj}) = \text{Map.empty}$
 ⟨proof⟩

lemma *update-slots-no-slots* [simp]:
 $\neg \text{has-slots } \text{obj} \implies \text{update-slots } \text{slots } \text{obj} = \text{obj}$
 ⟨proof⟩

lemma *update-slots-update-slots* [simp]:
 $\text{update-slots } \text{slots } (\text{update-slots } \text{slots}' \text{ obj}) = \text{update-slots } \text{slots } \text{obj}$
 ⟨proof⟩

lemma *update-slots-same-object*:
 $a = b \implies \text{update-slots } a \text{ obj} = \text{update-slots } b \text{ obj}$
 ⟨proof⟩

lemma *object-type-has-slots*:
 $\llbracket \text{has-slots } x; \text{object-type } x = \text{object-type } y \rrbracket \implies \text{has-slots } y$
 ⟨proof⟩

lemma *object-slots-object-clean-fields* [simp]:
 $\text{object-slots } (\text{object-clean-fields } \text{obj } \text{cmp}) = \text{object-slots } \text{obj}$
 ⟨proof⟩

lemma *object-slots-object-clean-slots* [simp]:
 $\text{object-slots } (\text{object-clean-slots } \text{obj } \text{cmp}) = \text{clean-slots } (\text{object-slots } \text{obj}) \text{ cmp}$
 ⟨proof⟩

lemma *object-slots-object-clean* [simp]:

$object-slots (object-clean\ obj\ cmp) = clean-slots (object-slots\ obj)\ cmp$

$\langle proof \rangle$

lemma *object-slots-add-to-slots* [simp]:

$object-type\ y = object-type\ z \implies object-slots (add-to-slots (object-slots\ y)\ z) = object-slots\ y ++ object-slots\ z$

$\langle proof \rangle$

lemma *update-slots-object-clean-slots* [simp]:

$update-slots\ slots (object-clean-slots\ obj\ cmp) = update-slots\ slots\ obj$

$\langle proof \rangle$

lemma *object-clean-fields-idem* [simp]:

$object-clean-fields (object-clean-fields\ obj\ cmp)\ cmp = object-clean-fields\ obj\ cmp$

$\langle proof \rangle$

lemma *object-clean-slots-idem* [simp]:

$object-clean-slots (object-clean-slots\ obj\ cmp)\ cmp = object-clean-slots\ obj\ cmp$

$\langle proof \rangle$

lemma *object-clean-fields-object-clean-slots* [simp]:

$object-clean-fields (object-clean-slots\ obj\ gs)\ gs = object-clean-slots (object-clean-fields\ obj\ gs)\ gs$

$\langle proof \rangle$

lemma *object-clean-idem* [simp]:

$object-clean (object-clean\ obj\ cmp)\ cmp = object-clean\ obj\ cmp$

$\langle proof \rangle$

lemma *has-slots-object-clean-slots*:

$has-slots (object-clean-slots\ obj\ cmp) = has-slots\ obj$

$\langle proof \rangle$

lemma *has-slots-object-clean-fields*:

$has-slots (object-clean-fields\ obj\ cmp) = has-slots\ obj$

$\langle proof \rangle$

lemma *has-slots-object-clean*:

$has-slots (object-clean\ obj\ cmp) = has-slots\ obj$

$\langle proof \rangle$

lemma *object-slots-update-slots-object-clean-fields* [simp]:

$object-slots (update-slots\ slots (object-clean-fields\ obj\ cmp)) = object-slots (update-slots\ slots\ obj)$

$\langle proof \rangle$

lemma *object-clean-fields-update-slots* [simp]:

$object-clean-fields (update-slots\ slots\ obj)\ cmp = update-slots\ slots (object-clean-fields\ obj)\ cmp$

obj cmp)
⟨*proof*⟩

lemma *object-clean-fields-twice* [*simp*]:

$(\text{object-clean-fields } (\text{object-clean-fields } \text{obj } \text{cmp}') \text{ cmp}) = \text{object-clean-fields } \text{obj}$
 $(\text{cmp} \cap \text{cmp}')$
⟨*proof*⟩

lemma *update-slots-object-clean-fields*:

$\llbracket \text{None} \notin \text{cmps}; \text{None} \notin \text{cmps}'; \text{object-type } \text{obj} = \text{object-type } \text{obj}' \rrbracket$
 $\implies \text{update-slots slots } (\text{object-clean-fields } \text{obj } \text{cmps}) =$
 $\text{update-slots slots } (\text{object-clean-fields } \text{obj}' \text{ cmps}')$
⟨*proof*⟩

lemma *object-clean-fields-no-slots*:

$\llbracket \text{None} \notin \text{cmps}; \text{None} \notin \text{cmps}'; \text{object-type } \text{obj} = \text{object-type } \text{obj}'; \neg \text{has-slots } \text{obj};$
 $\neg \text{has-slots } \text{obj}' \rrbracket$
 $\implies \text{object-clean-fields } \text{obj } \text{cmps} = \text{object-clean-fields } \text{obj}' \text{ cmps}'$
⟨*proof*⟩

lemma *update-slots-object-clean*:

$\llbracket \text{None} \notin \text{cmps}; \text{None} \notin \text{cmps}'; \text{object-type } \text{obj} = \text{object-type } \text{obj}' \rrbracket$
 $\implies \text{update-slots slots } (\text{object-clean } \text{obj } \text{cmps}) = \text{update-slots slots } (\text{object-clean}$
 $\text{obj}' \text{ cmps}')$
⟨*proof*⟩

lemma *cdl-heap-add-assoc'*:

$\forall \text{obj-id. not-conflicting-objects } x \ z \ \text{obj-id} \wedge$
 $\text{not-conflicting-objects } y \ z \ \text{obj-id} \wedge$
 $\text{not-conflicting-objects } x \ z \ \text{obj-id} \implies$
 $\text{cdl-heap-add } (\text{SepState } (\text{cdl-heap-add } x \ y) (\text{cdl-ghost-state-add } x \ y)) \ z =$
 $\text{cdl-heap-add } x \ (\text{SepState } (\text{cdl-heap-add } y \ z) (\text{cdl-ghost-state-add } y \ z))$
⟨*proof*⟩

lemma *cdl-heap-add-assoc*:

$\llbracket \text{sep-state-disj } x \ y; \text{sep-state-disj } y \ z; \text{sep-state-disj } x \ z \rrbracket$
 $\implies \text{cdl-heap-add } (\text{SepState } (\text{cdl-heap-add } x \ y) (\text{cdl-ghost-state-add } x \ y)) \ z =$
 $\text{cdl-heap-add } x \ (\text{SepState } (\text{cdl-heap-add } y \ z) (\text{cdl-ghost-state-add } y \ z))$
⟨*proof*⟩

lemma *cdl-ghost-state-add-assoc*:

$\text{cdl-ghost-state-add } (\text{SepState } (\text{cdl-heap-add } x \ y) (\text{cdl-ghost-state-add } x \ y)) \ z =$
 $\text{cdl-ghost-state-add } x \ (\text{SepState } (\text{cdl-heap-add } y \ z) (\text{cdl-ghost-state-add } y \ z))$
⟨*proof*⟩

lemma *clean-slots-map-add-comm*:

$\text{cmps-a} \cap \text{cmps-b} = \{\}$
 $\implies \text{clean-slots slots-a } \text{cmps-a} \ ++ \ \text{clean-slots slots-b } \text{cmps-b} =$
 $\text{clean-slots slots-b } \text{cmps-b} \ ++ \ \text{clean-slots slots-a } \text{cmps-a}$

$\langle \text{proof} \rangle$

lemma *object-clean-all*:

$\text{object-type } \text{obj-a} = \text{object-type } \text{obj-b} \implies \text{object-clean } \text{obj-b } \{\} = \text{object-clean } \text{obj-a } \{\}$
 $\langle \text{proof} \rangle$

lemma *object-add-comm*:

$\llbracket \text{object-type } \text{obj-a} = \text{object-type } \text{obj-b}; \text{cmps-a} \cap \text{cmps-b} = \{\} \rrbracket$
 $\implies \text{object-add } \text{obj-a } \text{obj-b } \text{cmps-a } \text{cmps-b} = \text{object-add } \text{obj-b } \text{obj-a } \text{cmps-b } \text{cmps-a}$
 $\langle \text{proof} \rangle$

lemma *sep-state-add-comm*:

$\text{sep-state-disj } x \ y \implies \text{sep-state-add } x \ y = \text{sep-state-add } y \ x$
 $\langle \text{proof} \rangle$

lemma *add-to-slots-comm*:

$\llbracket \text{object-slots } y\text{-obj} \perp \text{object-slots } z\text{-obj}; \text{update-slots } \text{Map.empty } y\text{-obj} = \text{update-slots } \text{Map.empty } z\text{-obj} \rrbracket$
 $\implies \text{add-to-slots } (\text{object-slots } z\text{-obj}) \ y\text{-obj} = \text{add-to-slots } (\text{object-slots } y\text{-obj}) \ z\text{-obj}$
 $\langle \text{proof} \rangle$

lemma *cdl-heap-add-none1*:

$\text{cdl-heap-add } x \ y \ \text{obj-id} = \text{None} \implies (\text{sep-heap } x) \ \text{obj-id} = \text{None}$
 $\langle \text{proof} \rangle$

lemma *cdl-heap-add-none2*:

$\text{cdl-heap-add } x \ y \ \text{obj-id} = \text{None} \implies (\text{sep-heap } y) \ \text{obj-id} = \text{None}$
 $\langle \text{proof} \rangle$

lemma *object-type-object-addL*:

$\text{object-type } \text{obj} = \text{object-type } \text{obj}'$
 $\implies \text{object-type } (\text{object-add } \text{obj } \text{obj}' \ \text{cmp } \text{cmp}') = \text{object-type } \text{obj}$
 $\langle \text{proof} \rangle$

lemma *object-type-object-addR*:

$\text{object-type } \text{obj} = \text{object-type } \text{obj}'$
 $\implies \text{object-type } (\text{object-add } \text{obj } \text{obj}' \ \text{cmp } \text{cmp}') = \text{object-type } \text{obj}'$
 $\langle \text{proof} \rangle$

lemma *sep-state-add-disjL*:

$\llbracket \text{sep-state-disj } y \ z; \text{sep-state-disj } x \ (\text{sep-state-add } y \ z) \rrbracket \implies \text{sep-state-disj } x \ y$
 $\langle \text{proof} \rangle$

lemma *sep-state-add-disjR*:

$\llbracket \text{sep-state-disj } y \ z; \text{sep-state-disj } x \ (\text{sep-state-add } y \ z) \rrbracket \implies \text{sep-state-disj } x \ z$
 $\langle \text{proof} \rangle$

lemma *sep-state-add-disj*:

```

[[sep-state-disj y z; sep-state-disj x y; sep-state-disj x z]] ==> sep-state-disj x
(sep-state-add y z)
⟨proof⟩

```

```

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

```

```

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

```

```

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

```

```

instance
  ⟨proof⟩

```

```

end

```

```

end

```

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

```

theory Separation-D
imports Abstract-Separation-D
begin

```

```

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

```

```

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

```

abbreviation

$$\text{lift}' :: (\text{sep-state} \Rightarrow 'a) \Rightarrow \text{cdl-state} \Rightarrow 'a \ (\langle \langle - \rangle \rangle)$$
where

$$\langle P \rangle s \equiv P \ (\text{state-sep-projection } s)$$
definition

$$\text{sep-map-general} :: \text{cdl-object-id} \Rightarrow \text{cdl-object} \Rightarrow \text{cdl-components} \Rightarrow \text{sep-pred}$$
where

$$\text{sep-map-general } p \ \text{obj } \text{gs} \equiv \lambda s. \text{sep-heap } s = [p \mapsto \text{obj}] \wedge \text{sep-ghost-state } s \ p = \text{gs}$$
lemma *sep-map-general-def2*:
$$\text{sep-map-general } p \ \text{obj } \text{gs} \ s =$$

$$(\text{dom } (\text{sep-heap } s) = \{p\} \wedge \text{ko-at } \text{obj } p \ (\text{sep-heap } s) \wedge \text{sep-ghost-state } s \ p = \text{gs})$$

$$\langle \text{proof} \rangle$$
definition

$$\text{sep-map-i} :: \text{cdl-object-id} \Rightarrow \text{cdl-object} \Rightarrow \text{sep-pred} \ (\langle - \mapsto i \rangle \ [76,71] \ 76)$$
where

$$p \mapsto i \ \text{obj} \equiv \text{sep-map-general } p \ \text{obj} \ \text{UNIV}$$
definition

$$\text{sep-map-f} :: \text{cdl-object-id} \Rightarrow \text{cdl-object} \Rightarrow \text{sep-pred} \ (\langle - \mapsto f \rangle \ [76,71] \ 76)$$
where

$$p \mapsto f \ \text{obj} \equiv \text{sep-map-general } p \ (\text{update-slots } \text{Map.empty} \ \text{obj}) \ \{\text{None}\}$$
definition

$$\text{sep-map-c} :: \text{cdl-cap-ref} \Rightarrow \text{cdl-cap} \Rightarrow \text{sep-pred} \ (\langle - \mapsto c \rangle \ [76,71] \ 76)$$
where

$$p \mapsto c \ \text{cap} \equiv \lambda s. \text{let } (\text{obj-id}, \ \text{slot}) = p; \ \text{heap} = \text{sep-heap } s \ \text{in}$$

$$\exists \ \text{obj}. \ \text{sep-map-general } \ \text{obj-id} \ \text{obj} \ \{\text{Some } \text{slot}\} \ s \wedge \ \text{object-slots } \ \text{obj} = [\text{slot} \mapsto \text{cap}]$$
definition

$$\text{sep-any} :: ('a \Rightarrow 'b \Rightarrow \text{sep-pred}) \Rightarrow ('a \Rightarrow \text{sep-pred}) \ \mathbf{where}$$

$$\text{sep-any } m \equiv (\lambda p \ s. \ \exists v. \ (m \ p \ v) \ s)$$
abbreviation *sep-any-map-i* $\equiv \text{sep-any } \text{sep-map-i}$ **notation** *sep-any-map-i* $(\langle - \mapsto i \rangle \ [76])$ **abbreviation** *sep-any-map-c* $\equiv \text{sep-any } \text{sep-map-c}$ **notation** *sep-any-map-c* $(\langle - \mapsto c \rangle \ [76])$ **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.