

Separation Algebra

Gerwin Klein and Rafal Kolanski and Andrew Boyton

December 14, 2021

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.	5
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>	8
4.6	Properties of zero (\square)	9
4.7	Properties of top (<i>sep-true</i>)	9
4.8	Separating Conjunction with Quantifiers	9
4.9	Properties of Separating Implication	10

4.10	Pure assertions	11
4.11	Intuitionistic assertions	12
4.12	Strictly exact assertions	15
5	Separation Algebra with Stronger, but More Intuitive Disjunction Axiom	16
6	Folding separating conjunction over lists of predicates	16
7	Separation Algebra with a Cancellative Monoid (for completeness)	17
8	Standard Heaps as an Instance of Separation Algebra	19
9	Separation Logic Tactics	21
10	Selection (move-to-front) tactics	21
11	Substitution	21
12	Forward Reasoning	22
13	Backward Reasoning	22
14	Cancellation of Common Conjuncts via Elimination Rules	22
15	Example from HOL/Hoare/Separation	23
16	Test cases for <i>sep-cancel</i>.	27
17	More properties of maps plus map disjunction.	28
18	Things that could go into Option Type	28
19	Things that go into Map.thy	28
19.1	Properties of maps not related to restriction	29
19.2	Properties of map restriction	30
20	Things that should not go into Map.thy (separation logic)	32
20.1	Definitions	32
20.2	Properties of ($'-$)	32
20.3	Properties of map disjunction	32
20.4	Map associativity-commutativity based on map disjunction . .	33
20.5	Basic properties	33
20.6	Map disjunction and addition	34
20.7	Map disjunction and map updates	36
20.8	Map disjunction and (\subseteq_m)	36

20.9 Map disjunction and restriction	37
21 Separation Algebra for Virtual Memory	39
22 Abstract Separation Logic, Alternative Definition	40
23 Equivalence between Separation Algebra Formulations	47
24 Total implies Partial	47
25 Partial implies Total	47
26 A simplified version of the actual capDL specification.	49
27 Instantiating capDL as a separation algebra.	53
28 Defining some separation logic maps-to predicates on top of the instantiation.	67

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

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:  $\llbracket x \#\# y + z; y \#\# z \rrbracket \implies x \#\# y$ 
  assumes sep-disj-addI1:  $\llbracket x \#\# y + z; y \#\# z \rrbracket \implies x + y \#\# z$ 
begin

```

4.1 Basic Construct Definitions and Abbreviations

definition

```

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

```

notation

```

sep-conj (infixr  $\wedge^*$  35)

```

definition

```

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

```

definition

```

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

```

definition

```

sep-substate :: 'a  $\Rightarrow$  'a  $\Rightarrow$  bool (infixr  $\preceq$  60) where
   $x \preceq y \equiv \exists z. x \#\# z \wedge x + z = y$ 

```

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) ( $\bigwedge^*$  - [60] 90) where
  sep-list-conj Ps  $\equiv foldl (**) \square Ps$ 

```

4.2 Disjunction/Addition Properties

lemma disjoint-zero-sym [simp]: $0 \#\# x$
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*: $\llbracket x \#\# y + z; y \#\# z \rrbracket \Longrightarrow x \#\# z$
by (*metis sep-disj-addD1 sep-add-ac*)

lemma *sep-disj-addD*: $\llbracket x \#\# y + z; y \#\# z \rrbracket \Longrightarrow x \#\# y \wedge x \#\# z$
by (*metis sep-disj-addD1 sep-disj-addD2*)

lemma *sep-add-disjD*: $\llbracket x + y \#\# z; x \#\# y \rrbracket \Longrightarrow x \#\# z \wedge y \#\# z$
by (*metis sep-disj-addD sep-disj-commuteI*)

lemma *sep-disj-addI2*:
 $\llbracket x \#\# y + z; y \#\# z \rrbracket \Longrightarrow x + z \#\# y$
by (*metis sep-add-ac sep-disj-addI1*)

lemma *sep-add-disjI1*:
 $\llbracket x + y \#\# z; x \#\# y \rrbracket \Longrightarrow x + z \#\# y$
by (*metis sep-add-ac sep-add-disjD sep-disj-addI2*)

lemma *sep-add-disjI2*:
 $\llbracket x + y \#\# z; x \#\# y \rrbracket \Longrightarrow z + y \#\# x$
by (*metis sep-add-ac sep-add-disjD sep-disj-addI2*)

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

lemma *sep-disj-add*:
 $\llbracket y \#\# z; x \#\# y \rrbracket \Longrightarrow x \#\# y + z = x + y \#\# z$
by (*metis sep-disj-addI1 sep-disj-addI3*)

4.3 Substate Properties

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

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

4.4 Separating Conjunction Properties

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

lemma *sep-conjE*:
 $\llbracket (P ** Q) h; \bigwedge x y. \llbracket P x; Q y; x \#\# y; h = x + y \rrbracket \Longrightarrow X \rrbracket \Longrightarrow X$
by (*auto simp: sep-conj-def*)

lemma *sep-conjI*:
 $\llbracket P\ x; Q\ y; x\ \#\#\ y; h = x + y \rrbracket \implies (P\ **\ Q)\ h$
by (*auto simp: sep-conj-def*)

lemma *sep-conj-commuteI*:
 $(P\ **\ Q)\ h \implies (Q\ **\ P)\ h$
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*:
 $\llbracket (P\ **\ Q)\ h; \bigwedge h. P\ h \implies P'\ h; \bigwedge h. Q\ h \implies Q'\ h \rrbracket \implies (P'\ **\ Q')\ h$
by (*erule sep-conjE, auto intro!: sep-conjI*)

lemma *sep-conj-impl1*:
assumes *P: \bigwedge h. P\ h \implies I\ h*
shows $(P\ **\ R)\ h \implies (I\ **\ R)\ h$
by (*auto intro: sep-conj-impl P*)

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

by (fast elim: sep-conj-impl)

lemma *sep-conj-trivial-strip2*:

$Q = R \implies (Q ** P) = (R ** P)$ by *simp*

lemma *disjoint-subheaps-exist*:

$\exists x y. x \#\# y \wedge h = x + y$

by (rule-tac $x=0$ in *exI*, *auto*)

lemma *sep-conj-left-commute*:

$(P ** (Q ** R)) = (Q ** (P ** R))$ (is $?x = ?y$)

proof –

have $?x = ((Q ** R) ** P)$ by (*simp add: sep-conj-commute*)

also have $\dots = (Q ** (R ** P))$ 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!*]: $\square 0$

by (*simp add: sep-empty-def*)

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

lemma *sep-conj-sep-true*:

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

by (*simp add: sep-conjI*[**where** $y=0$])

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

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

by (*simp add: sep-conjI*[**where** $x=0$])

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

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

unfolding *sep-conj-def*

by (*auto intro!: ext intro: disjoint-subheaps-exist*)

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

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

by (*force elim: sep-conjE intro!: ext*)

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

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

by (*subst sep-conj-commute*) (*rule sep-conj-false-right*)

4.6 Properties of zero (\square)

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

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

by (*simp add: sep-conj-def sep-empty-def*)

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

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

by (*subst sep-conj-commute, rule sep-conj-empty*)

lemma *sep-conj-sep-emptyI*:

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

by *simp*

lemma *sep-conj-sep-emptyE*:

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

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*]:

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

by (*simp add: sep-conj-assoc[symmetric]*)

lemma *sep-conj-disj*:

$$((P \text{ or } Q) ** R) = ((P ** R) \text{ or } (Q ** R))$$

by (*auto simp: sep-conj-def intro!: ext*)

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

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

by (*erule sep-conj-impl, simp+*)

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

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

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*:

$$((P \text{ and } Q) ** R) h \implies ((P ** R) \text{ and } (Q ** R)) h$$

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

4.9 Properties of Separating Implication

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

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

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

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

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

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

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

lemma *sep-conj-sep-impl*:
 $\llbracket P\ h; \bigwedge h. (P ** Q)\ h \implies R\ h \rrbracket \implies (Q \longrightarrow* R)\ h$
proof (*rule sep-implI*)
fix $h'\ h$
assume $P\ h$ **and** $h \#\# h'$ **and** $Q\ h'$

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

lemma *sep-conj-sep-impl2*:
 $\llbracket (P ** Q) h; \bigwedge h. P h \implies (Q \longrightarrow* R) h \rrbracket \implies R h$
by (*force dest: sep-implD elim: sep-conjE*)

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

4.10 Pure assertions

definition

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

lemma *pure-sep-conj*:
 $\llbracket \text{pure } P; \text{pure } Q \rrbracket \implies \text{pure } (P \wedge* Q)$
by (*force simp: pure-split*)

lemma *pure-sep-impl*:
 $\llbracket \text{pure } P; \text{pure } Q \rrbracket \implies \text{pure } (P \longrightarrow* Q)$
by (*force simp: pure-split*)

lemma *pure-conj-sep-conj*:
 $\llbracket (P \text{ and } Q) h; \text{pure } P \vee \text{pure } Q \rrbracket \implies (P \wedge* Q) h$
by (*metis pure-def sep-add-zero sep-conjI sep-conj-commute sep-disj-zero*)

lemma *pure-sep-conj-conj*:
 $\llbracket (P \wedge* Q) h; \text{pure } P; \text{pure } Q \rrbracket \implies (P \text{ and } Q) h$
by (*force simp: pure-split*)

lemma *pure-conj-sep-conj-assoc*:
 $\text{pure } P \implies ((P \text{ and } Q) \wedge* R) = (P \text{ and } (Q \wedge* R))$

by (*auto simp: pure-split*)

lemma *pure-sep-impl-impl*:

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

by (*force simp: pure-split dest: sep-impl-sep-true-P*)

lemma *pure-impl-sep-impl*:

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

by (*force simp: pure-split*)

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

by (*rule ext, rule, rule, clarsimp elim!: sep-conjE*)

(*erule sep-conj-impl, auto*)

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

by (*simp add: conj-comms pure-conj-right*)

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

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

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

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

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

lemma *intuitionisticI*:

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

by (*unfold intuitionistic-def, fast*)

lemma *intuitionisticD*:

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

by (*unfold intuitionistic-def, fast*)

lemma *pure-intuitionistic*:

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

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

lemma *intuitionistic-conj*:

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

by (*force intro: intuitionisticI dest: intuitionisticD*)

lemma *intuitionistic-disj*:
 $\llbracket \text{intuitionistic } P; \text{intuitionistic } Q \rrbracket \implies \text{intuitionistic } (P \text{ or } Q)$
by (*force intro: intuitionisticI dest: intuitionisticD*)

lemma *intuitionistic-forall*:
 $(\bigwedge x. \text{intuitionistic } (P x)) \implies \text{intuitionistic } (\text{ALLS } x. P x)$
by (*force intro: intuitionisticI dest: intuitionisticD*)

lemma *intuitionistic-exists*:
 $(\bigwedge x. \text{intuitionistic } (P x)) \implies \text{intuitionistic } (\text{EXS } x. P x)$
by (*force intro: intuitionisticI dest: intuitionisticD*)

lemma *intuitionistic-sep-conj-sep-true*:
 $\text{intuitionistic } (\text{sep-true } \wedge^* P)$
proof (*rule intuitionisticI*)
fix $h h' r$
assume $a: (\text{sep-true } \wedge^* P) h$
then obtain $x y$ **where** $P: P y$ **and** $h: h = x + y$ **and** $xyd: x \#\# y$
by - (*drule sep-conjD, clarsimp*)
moreover assume $a2: h \preceq h'$
then obtain z **where** $h': h' = h + z$ **and** $hzd: h \#\# z$
by (*clarsimp simp: sep-substate-def*)

moreover have $(P \wedge^* \text{sep-true}) (y + (x + z))$
using $P h hzd xyd$
by (*metis sep-add-disjI1 sep-disj-commute sep-conjI*)
ultimately show $(\text{sep-true } \wedge^* P) h'$ **using** hzd
by (*auto simp: sep-conj-commute sep-add-ac dest!: sep-disj-addD*)
qed

lemma *intuitionistic-sep-impl-sep-true*:
 $\text{intuitionistic } (\text{sep-true } \longrightarrow^* P)$
proof (*rule intuitionisticI*)
fix $h h'$
assume $imp: (\text{sep-true } \longrightarrow^* P) h$ **and** $hh': h \preceq h'$

from hh' **obtain** z **where** $h': h' = h + z$ **and** $hzd: h \#\# z$
by (*clarsimp simp: sep-substate-def*)
show $(\text{sep-true } \longrightarrow^* P) h'$ **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 $ip: \text{intuitionistic } (P::('a \Rightarrow \text{bool}))$
shows $\text{intuitionistic } (P \wedge^* Q)$
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 $iq: \text{intuitionistic } Q$
shows $\text{intuitionistic } (P \longrightarrow^* Q)$
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*:
 $\neg (\exists Q. (\forall h. (Q h \longrightarrow (P \wedge^* \text{sep-true}) h)) \wedge \text{intuitionistic } Q \wedge Q \neq (P \wedge^* \text{sep-true}) \wedge (\forall h. P h \longrightarrow Q h))$

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

lemma *weakest-intuitionistic*:

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

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*:

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

by (*force dest: intuitionisticD elim: sep-conjE sep-substate-disj-add*)

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

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

by (*fast intro!: sep-conj-sep-true ext*)

elim: intuitionistic-sep-conj-sep-true-P)

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

$\llbracket P\ h; intuitionistic\ P \rrbracket \Longrightarrow (sep\text{-}true \longrightarrow* P)\ h$

by (*force intro!: sep-implI dest: intuitionisticD*)

intro: sep-substate-disj-add)

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

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

by (*fast intro!: ext*)

elim: sep-impl-sep-true-P intuitionistic-sep-impl-sep-true-P)

4.12 Strictly exact assertions

definition *strictly-exact* :: $('a \Rightarrow bool) \Rightarrow bool$ **where**

strictly-exact $P \equiv \forall h\ h'. P\ h \wedge P\ h' \longrightarrow h = h'$

lemma *strictly-exactD*:

$\llbracket strictly\text{-}exact\ P; P\ h; P\ h' \rrbracket \Longrightarrow h = h'$

by (*unfold strictly-exact-def, fast*)

lemma *strictly-exactI*:

$(\bigwedge h\ h'. \llbracket P\ h; P\ h' \rrbracket \Longrightarrow h = h') \Longrightarrow strictly\text{-}exact\ P$

by (*unfold strictly-exact-def, fast*)

lemma *strictly-exact-sep-conj*:

$\llbracket strictly\text{-}exact\ P; strictly\text{-}exact\ Q \rrbracket \Longrightarrow strictly\text{-}exact\ (P \wedge* Q)$

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  $\wedge^*$  sep-true) h; P h; strictly-exact Q ]]  $\impl$  (Q  $\wedge^*$  (Q  $\longrightarrow^*$  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* (**) \square
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*]: $y \#\# z \impl x \#\# y + z = (x \#\# y \wedge x \#\# z)$
begin

lemma *sep-disj-add-eq* [*simp*]: $x \#\# y \impl x + y \#\# z = (x \#\# z \wedge y \#\# z)$
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*]: $\bigwedge^* [] = \square$
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 $x + (\text{foldl } (+) \ 0 \ zs) = \text{foldl } (+) \ x \ zs$
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 ← xs . ¬ P x*])
by (*simp only: eq-commute*)

have *foldl-Cons'*:

$\bigwedge x xs. \text{foldl } (+) \ 0 \ (x \# \ xs) = x + (\text{foldl } (+) \ 0 \ xs)$
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*: $\llbracket x + z = y + z ; x \#\# z ; y \#\# z \rrbracket \implies x = y$

begin

definition

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

$precise\ P = (\forall h\ hp\ hp'.\ hp \preceq h \wedge P\ hp \wedge hp' \preceq h \wedge P\ hp' \longrightarrow hp = hp')$

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

lemma *sep-add-cancel*:
 $x\ \#\#\ z \implies y\ \#\#\ z \implies (x + z = y + z) = (x = y)$
by (*metis* *sep-add-cancelD*)

lemma *precise-distribute*:
 $precise\ P = (\forall Q\ R.\ ((Q\ and\ R) \wedge* P) = ((Q \wedge* P) and (R \wedge* P)))$

proof (*rule iffI*)

assume *pp*: *precise P*

{
fix *Q R*
fix *h hp hp' s*

{ **assume** *a*: $((Q\ and\ R) \wedge* P)\ s$
hence $((Q \wedge* P) and (R \wedge* P))\ s$
by (*fastforce* *dest!*: *sep-conjD* *elim*: *sep-conjI*)

}

moreover

{ **assume** *qs*: $(Q \wedge* P)\ s$ **and** *qr*: $(R \wedge* P)\ s$

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*: $y \preceq x + y$ **using** *xy*

by (*fastforce* *simp*: *sep-substate-disj-add'* *sep-disj-commute*)

from *sxy'* **have** *ys'*: $y' \preceq x' + y'$ **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 $((Q\ and\ R) \wedge* P)\ s$ **using** *sxy x x' yy y' xy'*

by (*fastforce* *intro*: *sep-conjI*)

}

ultimately

have $((Q\ and\ R) \wedge* P)\ s = ((Q \wedge* P) and (R \wedge* P))\ s$ **using** *pp* **by** *blast*

}

thus $\forall Q\ R.\ ((Q\ and\ R) \wedge* P) = ((Q \wedge* P) and (R \wedge* P))$ **by** (*blast* *intro!*: *ext*)

```

next
assume a:  $\forall Q R. ((Q \text{ and } R) \wedge* P) = ((Q \wedge* P) \text{ and } (R \wedge* P))$ 
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:  $P \text{ hp}$  and php':  $P \text{ hp}'$ 

  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  $\forall Q R. ((Q \text{ and } R) \wedge* P) (z + hp) = ((Q \wedge* P) \text{ and } (R \wedge* P)) (z' + hp')$ 
  by (fastforce simp: h-eq sep-add-ac sep-conj-commute)

  hence  $((=(=) z \text{ and } (=) z') \wedge* P) (z + hp) =$ 
     $((=(=) z \wedge* P) \text{ and } ((=) z' \wedge* P)) (z' + hp')$  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  $\implies$  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 cancella-
  tion 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* \Rightarrow *nat* \Rightarrow *heap* \Rightarrow *bool* ($- \mapsto -$ [56,51] 56)
where $x \mapsto y \equiv \lambda h. h = [x \mapsto y]$

notation *pred-ex* (**binder** \exists 10)

definition *maps-to-ex* :: *nat* \Rightarrow *heap* \Rightarrow *bool* ($- \mapsto -$ [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*:
 $(p \mapsto - ** P) H \Longrightarrow (p \mapsto v ** P) (H (p \mapsto v))$
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*:
 $(p \mapsto v ** P) H \Longrightarrow \text{the } (H p) = v$
by (*auto elim!: sep-conjE*
simp: sep-disj-fun-def maps-to-def map-convs map-add-def
split: option.splits)

primrec
 $list :: nat \Rightarrow nat\ list \Rightarrow heap \Rightarrow bool$
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 $list\ 0\ xs = (\lambda s. xs = [] \wedge \square s)$
by (*cases xs auto*)

lemma *VARs* $x\ y\ z\ w\ h$
 $\{(x \mapsto y ** z \mapsto w)\ h\}$
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
 $\{p \neq 0 \wedge (p \mapsto - **\ list\ q\ qs)\ H\}$
 $H := H(p \mapsto q)$
 $\{list\ p\ (p \# qs)\ H\}$
 apply *vcg*
 apply (*auto intro: maps-to-write*)
done

lemma *VARs* $H\ p\ q\ r$
 $\{(list\ p\ Ps\ **\ list\ q\ Qs)\ H\}$
 WHILE $p \neq 0$
 INV $\{\exists ps\ qs.\ (list\ p\ ps\ **\ list\ q\ qs)\ H \wedge rev\ ps\ @\ qs = rev\ Ps\ @\ Qs\}$
 DO $r := p; p := the(H\ p); H := H(r \mapsto q); q := r\ OD$
 $\{list\ q\ (rev\ Ps\ @\ Qs)\ H\}$
 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* \langle resolve-tac @{context} [mk-sep-select-rule @{context} false (4, 2)]
1 \rangle)

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 \llbracket also unrelated; ($A ** B ** Q ** P$) s $\rrbracket \implies$ unrelated
apply (*sep-select-asm* 2)
oops

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 $\bigwedge xv yv P s y x s. (A g x yv ** A g y yv ** P) s \implies (AA g y ** sep-true) s$
by (*sep-cancel add: forward*)

lemma
assumes *forward*: $\bigwedge s. generic s \implies instance s$
shows ($A ** generic ** B$) $s \implies (instance ** sep-true) s$
by (*sep-cancel add: forward*)

lemma $\llbracket (A ** B) sa ; (A ** Y) s \rrbracket \implies (A ** X) s$
apply (*sep-cancel*)
oops

lemma $\llbracket (A ** B) sa ; (A ** Y) s \rrbracket \implies (\lambda s. (A ** X) s) s$
apply (*sep-cancel*)
oops

schematic-goal $\llbracket (B ** A ** C) s \rrbracket \implies (\lambda s. (A ** ?X) s) s$
by (*sep-cancel*)

lemma
assumes *forward*: $\bigwedge s. generic s \implies instance s$
shows $\llbracket (A ** B) s ; (generic ** Y) s \rrbracket \implies (X ** instance) s$
apply (*sep-cancel add: forward*)
oops

lemma

assumes *forward*: $\bigwedge s. \text{generic } s \implies \text{instance } s$
shows *generic* $s \implies \text{instance } s$
by (*sep-cancel add: forward*)

lemma

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

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

lemma *None-com*: $(\text{None} = x) = (x = \text{None})$ **by** *fast*

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

19 Things that go into Map.thy

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

definition

map-inter :: $('a \rightarrow 'b) \Rightarrow ('a \rightarrow 'b) \Rightarrow 'a \text{ set}$ (**infixl** \cap_m 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 :: $('a \rightarrow 'b) \Rightarrow 'a \text{ set} \Rightarrow ('a \rightarrow 'b)$ (**infixl** $'-$ 110)
where
 $m \text{ '}- } S \equiv (\lambda x. \text{if } x \in S \text{ then } \text{None} \text{ else } m x)$

19.1 Properties of maps not related to restriction

lemma *empty-forall-equiv*: $(m = \text{Map.empty}) = (\forall x. m\ x = \text{None})$
by (*fastforce intro!*: *ext*)

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

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

lemma *non-dom-eval*:
 $x \notin \text{dom } m \implies m\ x = \text{None}$
by *auto*

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

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

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

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

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

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

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*:
 $\llbracket m_0 \subseteq_m m_1 ; \text{dom } m_0 = \text{dom } m_1 \rrbracket \implies m_0 = m_1$
by (*auto intro!: ext simp: map-le-def elim!: ballE*)

19.2 Properties of map restriction

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

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

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

lemma *restrict-map-UNIV* [*simp*]:
 $m \upharpoonright \text{UNIV} = m$
by (*simp add: restrict-map-def*)

lemma *restrict-map-dom*:
 $S = \text{dom } m \implies m \upharpoonright S = m$
by (*auto intro!: ext simp: restrict-map-def None-not-eq*)

lemma *restrict-map-subdom*:
 $\text{dom } m \subseteq S \implies m \upharpoonright S = m$
by (*fastforce simp: restrict-map-def None-com intro: ext*)

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

lemma *map-le-restrict*:
 $m \subseteq_m m' \implies m = m' \upharpoonright \text{dom } m$
by (*force simp: map-le-def restrict-map-def None-com intro: ext*)

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

lemma *restrict-map-remerge*:

$\llbracket S \cap T = \{\} \rrbracket \implies m \upharpoonright' S ++ m \upharpoonright' T = m \upharpoonright' (S \cup T)$
by (*rule ext, clarsimp simp: restrict-map-def map-add-def*
split: option.splits)

lemma *restrict-map-empty*:

$\text{dom } m \cap S = \{\} \implies m \upharpoonright' S = \text{Map.empty}$
by (*fastforce simp: restrict-map-def intro: ext*)

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

$(m \upharpoonright' S ++ m \upharpoonright' (\text{UNIV} - S)) = m$
by (*force simp: map-add-def restrict-map-def split: option.splits intro: ext*)

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

$(m \upharpoonright' S ++ m \upharpoonright' (\text{dom } m - S)) = m$
by (*auto simp: map-add-def restrict-map-def split: option.splits intro!: ext*)

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

$(m \upharpoonright' (\text{UNIV} - S) ++ m \upharpoonright' S) = m$
by (*subst map-add-comm, auto*)

lemma *restrict-self-UNIV*:

$m \upharpoonright' (\text{dom } m - S) = m \upharpoonright' (\text{UNIV} - S)$
by (*auto intro!: ext simp: restrict-map-def*)

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

$x \notin \text{dom } m' \implies (m ++ m') \upharpoonright' \{x\} = m \upharpoonright' \{x\}$
by (*rule ext, auto simp: restrict-map-def map-add-def split: option.splits*)

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

$x \notin \text{dom } m \implies (m ++ m') \upharpoonright' \{x\} = m' \upharpoonright' \{x\}$
by (*rule ext, auto simp: restrict-map-def map-add-def split: option.splits*)

lemma *map-add-restrict-right*:

$x \subseteq \text{dom } m' \implies (m ++ m') \upharpoonright' x = m' \upharpoonright' x$
by (*rule ext, auto simp: restrict-map-def map-add-def split: option.splits*)

lemma *restrict-map-compose*:

$\llbracket S \cup T = \text{dom } m ; S \cap T = \{\} \rrbracket \implies m \upharpoonright' S ++ m \upharpoonright' T = m$
by (*fastforce intro: ext simp: map-add-def restrict-map-def*)

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

$\llbracket m' \subseteq_m m ; \text{dom } m' \subseteq S \rrbracket \implies m' \subseteq_m (m \upharpoonright' S)$
by (*force simp: restrict-map-def map-le-def*)

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

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

$T \subseteq S \implies m \upharpoonright (S - T) ++ m \upharpoonright T = m \upharpoonright S$
by (*auto simp: restrict-map-def map-add-def intro!: ext split: option.splits*)

lemma *restrict-map-sub-union*:

$m \upharpoonright (dom\ m - (S \cup T)) = (m \upharpoonright (dom\ m - T)) \upharpoonright (dom\ m - S)$
by (*auto intro!: ext simp: restrict-map-def*)

lemma *prod-restrict-map-add*:

$\llbracket S \cup T = U; S \cap T = \{\} \rrbracket \implies m \upharpoonright (X \times S) ++ m \upharpoonright (X \times T) = m \upharpoonright (X \times U)$
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 disjunction

definition

$map\text{-}disj :: ('a \multimap 'b) \Rightarrow ('a \multimap 'b) \Rightarrow bool$ (**infix** \perp 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*: $h \upharpoonright S \perp h \upharpoonright S$

by (*fastforce simp: sub-restrict-map-def restrict-map-def map-disj-def split: option.splits if-split-asm*)

lemma *restrict-map-sub-add*: $h \upharpoonright S ++ h \upharpoonright S = h$

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

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

20.4 Map associativity-commutativity based on map disjunction

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

lemma *map-add-left-commute*:
 $h_0 \perp h_1 \implies h_0 ++ (h_1 ++ h_2) = h_1 ++ (h_0 ++ h_2)$
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*:
 $\llbracket h_0 \perp h_1 ; x \in \text{dom } h_0 \rrbracket \implies h_1 x = \text{None}$
by (*auto simp: map-disj-def dom-def*)

lemma *map-disj-None-left*:
 $\llbracket h_0 \perp h_1 ; x \in \text{dom } h_1 \rrbracket \implies h_0 x = \text{None}$
by (*auto simp: map-disj-def dom-def*)

lemma *map-disj-None-left'*:
 $\llbracket h_0 x = \text{Some } y ; h_1 \perp h_0 \rrbracket \implies h_1 x = \text{None}$
by (*auto simp: map-disj-def*)

lemma *map-disj-None-right'*:
 $\llbracket h_1 x = \text{Some } y ; h_1 \perp h_0 \rrbracket \implies h_0 x = \text{None}$
by (*auto simp: map-disj-def*)

lemma *map-disj-common*:
 $\llbracket h_0 \perp h_1 ; h_0 p = \text{Some } v ; h_1 p = \text{Some } v' \rrbracket \implies \text{False}$
by (*frule (1) map-disj-None-left', simp*)

lemma *map-disj-eq-dom-left*:
 $\llbracket h_0 \perp h_1 ; \text{dom } h_0' = \text{dom } h_0 \rrbracket \implies h_0' \perp h_1$
by (*auto simp: map-disj-def*)

20.6 Map disjunction and addition

lemma *map-add-eval-left*:
 $\llbracket x \in \text{dom } h ; h \perp h' \rrbracket \implies (h ++ h') x = h x$
by (*auto dest!: map-disj-None-right simp: map-add-def cong: option.case-cong*)

lemma *map-add-eval-right*:
 $\llbracket x \in \text{dom } h' ; h \perp h' \rrbracket \implies (h ++ h') x = h' x$
by (*auto elim!: map-disjD simp: map-add-comm map-add-eval-left map-disj-com*)

lemma *map-add-eval-left'*:
 $\llbracket x \notin \text{dom } h' ; h \perp h' \rrbracket \implies (h ++ h') x = h x$
by (*clarsimp simp: map-disj-def map-add-def split: option.splits*)

lemma *map-add-eval-right'*:
 $\llbracket x \notin \text{dom } h ; h \perp h' \rrbracket \implies (h ++ h') x = h' x$
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*: $h_0 \perp h_1 \ h_0' \perp h_1' \ \text{dom } h_0 = \text{dom } h_0'$
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** $x \in \text{dom } h$
hence $h_0 x = h_1 x$ **using** *disj* **by** (*simp add: map-disj-None-left*)
} **moreover** **{**
assume $x \notin \text{dom } h$
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*:
 $\llbracket h ++ h_0 = h ++ h_1; h_0 \perp h; h_1 \perp h \rrbracket \implies h_0 = h_1$
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** $d: \text{dom } h_0 = \text{dom } h_0'$ **and**
ab-disj: $h_0 \perp h_1$ **and** *cd-disj*: $h_0' \perp h_1'$
shows $h_1 = h_1'$
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: $\text{dom } h_1 = \text{dom } h_1'$ **and**
disj: $h_0 \perp h_1$ $h_0' \perp h_1'$
shows $h_0 = h_0'$
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** $(\text{auto intro!}: \text{ext})$
hence $h_1 x = h_1' x$ **using** disj
by $- (\text{cases } x \in \text{dom } h_0,$
 $\quad \text{simp-all add: map-disj-None-right map-add-eval-right}')$
thus $h_1 x = h_1' x$ **by** $(\text{auto intro!}: \text{ext})$
qed auto

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

20.7 Map disjunction and map updates

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

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

lemma $\text{map-add-update-left}$:
 $\llbracket h_0 \perp h_1 ; p \in \text{dom } h_0 \rrbracket \implies (h_0 ++ h_1)(p \mapsto v) = (h_0(p \mapsto v) ++ h_1)$
by $(\text{drule } (1) \text{ map-disj-None-right})$
 $(\text{auto intro: ext simp: map-add-def cong: option.case-cong})$

lemma $\text{map-add-update-right}$:
 $\llbracket h_0 \perp h_1 ; p \in \text{dom } h_1 \rrbracket \implies (h_0 ++ h_1)(p \mapsto v) = (h_0 ++ h_1(p \mapsto v))$
by $(\text{drule } (1) \text{ map-disj-None-left})$
 $(\text{auto intro: ext simp: map-add-def cong: option.case-cong})$

lemma map-add3-update :
 $\llbracket h_0 \perp h_1 ; h_1 \perp h_2 ; h_0 \perp h_2 ; p \in \text{dom } h_0 \rrbracket$
 $\implies (h_0 ++ h_1 ++ h_2)(p \mapsto v) = h_0(p \mapsto v) ++ h_1 ++ h_2$
by $(\text{auto simp: map-add-update-left[symmetric] map-add-ac})$

20.8 Map disjunction and (\subseteq_m)

lemma map-le-override $[\text{simp}]$:
 $\llbracket h \perp h' \rrbracket \implies h \subseteq_m h ++ h'$
by $(\text{auto simp: map-le-def map-add-def map-disj-def split: option.splits})$

lemma map-leI-left :
 $\llbracket h = h_0 ++ h_1 ; h_0 \perp h_1 \rrbracket \implies h_0 \subseteq_m h$ **by** auto

lemma map-leI-right :
 $\llbracket h = h_0 ++ h_1 ; h_0 \perp h_1 \rrbracket \implies h_1 \subseteq_m h$ **by** auto

lemma *map-disj-map-le*:

$\llbracket h_0' \subseteq_m h_0; h_0 \perp h_1 \rrbracket \implies h_0' \perp h_1$
by (*force simp: map-disj-def map-le-def*)

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

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

$\llbracket h' \subseteq_m h; h_0 \perp h_1; h' = h_1 ++ h_0 \rrbracket \implies h_0 \subseteq_m h$
by (*auto simp: map-le-on-disj-left map-add-ac*)

lemma *map-le-add-cancel*:

$\llbracket h_0 \perp h_1; h_0' \subseteq_m h_0 \rrbracket \implies h_0' ++ h_1 \subseteq_m h_0 ++ h_1$
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*: $a \in \text{dom } h_0'$
hence *sumeq*: $(h_0' ++ h_1) a = (h_0 ++ h_1) a$
using *subm* **unfolding** *map-le-def* **by** *auto*
from *a* **have** $a \notin \text{dom } h_1$ **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= $\lambda x. \text{if } x \notin \text{dom } h_0' \text{ then } h_0 \text{ else None}$])
(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*]:

$h_0 \perp h_1 \mid' (\text{UNIV} - \text{dom } h_0)$
by (*force simp: map-disj-def*)

lemma *restrict-map-disj*:

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

by (*auto simp: map-disj-def restrict-map-def dom-def*)

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

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

by (*force simp: map-disj-def*)

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

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

by (*fastforce simp: map-disj-def restrict-map-def intro: ext*)

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

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

by (*rule ext, auto simp: map-disj-def restrict-map-def*)

lemma *restrict-map-disj-dom*:

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

by (*auto simp: map-disj-def restrict-map-def dom-def*)

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

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

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'*:

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

by (*rule ext, auto simp: restrict-map-def map-add-def dom-def map-disj-def split: option.splits*)

lemma *restrict-map-disj-left*:

$$h_0 \perp h_1 \implies h_0 \mid' S \perp h_1$$

by (*auto simp: map-disj-def*)

lemma *restrict-map-disj-right*:

$$h_0 \perp h_1 \implies h_0 \perp h_1 \mid' S$$

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*:

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

by (*simp add: map-add-restrict restrict-map-empty map-disj-def*)

lemma *restrict-map-on-disj*:

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

unfolding *map-disj-def* **by** *auto*

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

$h_0 \perp h_1 \implies h_0 \perp h_1 \mid' S$
by (*auto simp: map-disj-def map-add-def*)

lemma *map-le-sub-dom*:

$\llbracket h_0 ++ h_1 \subseteq_m h ; h_0 \perp h_1 \rrbracket \implies h_0 \subseteq_m h \mid' (dom\ h - dom\ h_1)$
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*:

$\llbracket h \subseteq_m h' \rrbracket \implies h' = (h' \mid' (UNIV - dom\ h)) ++ h$
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*:

$\llbracket h_0 \perp h_1 ; S \cap S' = \{\}; T \cap T' = \{\} \rrbracket$
 $\implies (h_0 \mid' S) ++ (h_1 \mid' T) \perp (h_0 \mid' S') ++ (h_1 \mid' T')$
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*
 $= VM\text{SepState } ((('p \times 'v) \multimap 'value) \times 'r\ \text{set})$

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

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

□ ≡ λs. s = 0

definition

$sep_impl :: ('a \Rightarrow bool) \Rightarrow ('a \Rightarrow bool) \Rightarrow ('a \Rightarrow bool)$ (**infixr** \longrightarrow^* 25)
where
 $P \longrightarrow^* Q \equiv \lambda h. \forall h' h''. h \oplus h' = Some\ h'' \wedge P\ h' \longrightarrow Q\ h''$

definition (in -)

$sep_true \equiv \lambda s. True$

definition (in -)

$sep_false \equiv \lambda s. False$

abbreviation

$add2 :: 'a\ option \Rightarrow 'a\ option \Rightarrow 'a\ option$ (**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:

$Some\ x ++ Some\ y = x \oplus y$

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:

$\llbracket (P ** Q)\ s; \bigwedge p\ q. \llbracket P\ p; Q\ q; p \oplus q = Some\ s \rrbracket \Longrightarrow X \rrbracket \Longrightarrow X$

by (*auto simp: sep-conj-def*)

lemma sep-conjI:

$\llbracket P p; Q q; p \oplus q = \text{Some } s \rrbracket \implies (P ** Q) s$
by (*auto simp: sep-conj-def*)

lemma *sep-conj-comI*:
 $(P ** Q) s \implies (Q ** P) s$
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*:
 $\llbracket z \oplus q = \text{Some } s; x \oplus y = \text{Some } q \rrbracket \implies \text{Some } z ++ \text{Some } x ++ \text{Some } y = \text{Some } s$
by (*simp add: add2-Some-Some*)

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

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

lemma *Some-add2*:
 $(\text{Some } x ++ y = \text{Some } z) = (\exists y'. y = \text{Some } y' \wedge x \oplus y' = \text{Some } z)$
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]*: $\neg \text{sep-false } x$ **by** (*simp add: sep-false-def*)

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

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

lemma *sep-conj-sep-true'*:
 $P s \implies (sep\text{-}true ** P) s$
by (auto simp: sep-conjI [where p=0])

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

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

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

lemma *sep-conj-false-left[simp]*:
 $(sep\text{-}false ** P) = sep\text{-}false$
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]*: $\square 0$
by (simp add: emp-def)

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

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

lemma *sep-conj-sep-emptyI*:
 $P s \implies (P ** \square) s$
by *simp*

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

$(sep\text{-}true ** (sep\text{-}true ** P)) = (sep\text{-}true ** P)$
by (*simp add: sep-conj-assoc*)

lemma *sep-conj-disj*:

$((\lambda s. P s \vee Q s) ** R) s = ((P ** R) s \vee (Q ** R) s)$ (**is** $?x = (?y \vee ?z)$)
by (*auto simp: sep-conj-def*)

lemma *sep-conj-conj*:

$((\lambda s. P s \wedge Q s) ** R) s \implies (P ** R) s \wedge (Q ** R) s$
by (*force intro: sep-conjI elim!: sep-conjE*)

lemma *sep-conj-exists1*:

$((\lambda s. \exists x. P x s) ** Q) s = (\exists x. (P x ** Q) s)$
by (*force intro: sep-conjI elim: sep-conjE*)

lemma *sep-conj-exists2*:

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

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

lemma *sep-conj-forall*:

$((\lambda s. \forall x. P x s) ** Q) s \implies (P x ** Q) s$
by (*force intro: sep-conjI elim: sep-conjE*)

lemma *sep-conj-impl*:

$\llbracket (P ** Q) s; \bigwedge s. P s \implies P' s; \bigwedge s. Q s \implies Q' s \rrbracket \implies (P' ** Q') s$
by (*erule sep-conjE, auto intro!: sep-conjI*)

lemma *sep-conj-impl1*:

assumes $P: \bigwedge s. P s \implies I s$
shows $(P ** R) s \implies (I ** R) s$
by (*auto intro: sep-conj-impl P*)

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

$(P ** Q) s \implies (sep\text{-}true ** Q) s$
by (*erule sep-conj-impl, simp+*)

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

$(P ** Q) s \implies (P ** sep\text{-}true) s$
by (*subst (asm) sep-conj-com, drule sep-conj-sep-true-left, simp add: sep-conj-ac*)

lemma *sep-globalise*:

$\llbracket (P ** R) s; \bigwedge s. P s \implies Q s \rrbracket \implies (Q ** R) s$
by (*fast elim: sep-conj-impl*)

lemma *sep-implI*:

assumes $a: \bigwedge h' h''. \llbracket h \oplus h' = \text{Some } h''; P h' \rrbracket \implies Q h''$

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

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

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

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

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

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

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

lemma *sep-conj-sep-impl2*:
 $\llbracket (P ** Q) s; \bigwedge s. P s \implies (Q \longrightarrow^* R) s \rrbracket \implies R s$
 by (*force dest: sep-implD elim: sep-conjE*)

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

lemma *sep-conj-triv-strip2*:
 $Q = R \implies (Q ** P) = (R ** P)$ 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* \equiv 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 \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
by *simp*

lemma *sep-disj-commuteI'*:
disjoint *x y* \implies *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  $\implies$  sep-add' x y = sep-add' y x
by (clarsimp simp: sep-add'-def disjoint-def add-comm)

lemma sep-add-assoc':
   $\llbracket$  disjoint x y; disjoint y z; disjoint x z  $\rrbracket \implies$ 
  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)  $\implies$  disjoint y z  $\implies$  disjoint x y
proof (clarsimp simp: disjoint-def sep-add'-def)
  fix a assume a: y  $\oplus$  z = Some a
  fix b assume b: x  $\oplus$  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  $\exists$  b. x  $\oplus$  y = Some b by (simp add: lift2-def split: option.splits)
qed

lemma sep-disj-addI1':
  disjoint x (sep-add' y z)  $\implies$  disjoint y z  $\implies$  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  $\times$  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 $cap\text{-objects} :: cdl\text{-cap} \Rightarrow cdl\text{-object-id set}$

where

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

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

where

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

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

where

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

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

$cap\text{-object (TcbCap x)} = x$
 $cap\text{-object (CNodeCap x)} = x$
 $cap\text{-object (EndpointCap x j)} = x$
by ($simp\text{-all add:cap-object-def cap-objects-def cap-has-object-def}$)

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

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

where

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

definition

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

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:
 $\llbracket x \in A; A \cap B = \{\} \rrbracket \implies x \notin B$ 
  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':
 $S \cap T = \{\} \implies h \mid' S \perp h' \mid' T$ 
  by (auto simp: map-disj-def restrict-map-def dom-def)

lemma map-add-restrict-comm:
 $S \cap T = \{\} \implies h \mid' S ++ h' \mid' T = h' \mid' T ++ h \mid' S$ 
  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  $\Rightarrow$  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-set :: 'a option set \Rightarrow 'a set

where

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

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

the-set (A \cup B) = *the-set* A \cup *the-set* B

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

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

the-set (A \cap B) = *the-set* A \cap *the-set* B

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

lemma *the-set-inter-empty*:

A \cap B = {} \implies *the-set* A \cap *the-set* B = {}

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

definition

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

where

slots-of-heap h \equiv λ obj-id.

case h obj-id of

None \Rightarrow Map.empty

| Some obj \Rightarrow object-slots 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

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 \implies 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 \implies 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 \implies 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 \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* \Rightarrow CNode (*x*(\|*cdl-cnode-size-bits* := *undefined* \|))
 | - \Rightarrow *obj*

definition

object-clean-slots :: *cdl-object* \Rightarrow *cdl-components* \Rightarrow *cdl-object*

where

object-clean-slots *obj cmp* \equiv *update-slots* (*clean-slots* (*object-slots* *obj*) *cmp*) *obj*

definition

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

where

object-clean *obj gs* \equiv *object-clean-slots* (*object-clean-fields* *obj gs*) *gs*

definition

object-add :: *cdl-object* \Rightarrow *cdl-object* \Rightarrow *cdl-components* \Rightarrow *cdl-components* \Rightarrow *cdl-object*

where

object-add *obj-a obj-b cmps-a cmps-b* \equiv
 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* \in *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* \Rightarrow *sep-state* \Rightarrow *cdl-heap*

where

cdl-heap-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 case *heap-b obj-id* of
 None \Rightarrow *heap-a obj-id*

| *Some obj-b* \Rightarrow *case heap-a obj-id of*
 None \Rightarrow *heap-b obj-id*
 | *Some obj-a* \Rightarrow *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*

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*:
 $\llbracket \text{object-type } obj\text{-}a = \text{object-type } obj\text{-}b;$
 $\text{object-slots } obj\text{-}a ++ \text{object-slots } obj\text{-}b = \text{object-slots } obj\text{-}b ++ \text{object-slots } obj\text{-}a;$
 $None \notin cmp \rrbracket$
 $\implies \text{object-clean } (\text{add-to-slots } (\text{object-slots } obj\text{-}a) obj\text{-}b) cmp =$
 $\text{object-clean } (\text{add-to-slots } (\text{object-slots } obj\text{-}b) obj\text{-}a) cmp$
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*:
 $\text{object-type } y = \text{object-type } z$
 $\implies \text{add-to-slots } (\text{object-slots } (\text{add-to-slots } (x) y)) z =$
 $\text{add-to-slots } (x ++ \text{object-slots } y) z$
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*]:
 $\text{not-conflicting-objects } s (\text{SepState Map.empty } (\lambda obj\text{-}id. \{\})) obj\text{-}id$
by (*clarsimp simp: not-conflicting-objects-def split:option.splits*)

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

lemma *not-conflicting-objects-empty-object* [*elim!*]:
 $(\text{sep-heap } x) obj\text{-}id = None \implies \text{not-conflicting-objects } x y obj\text{-}id$
by (*clarsimp simp: not-conflicting-objects-def*)

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

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

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

```

cdl-heap-add (SepState Map.empty ( $\lambda$ 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:  $a ++ b = \text{Map.empty} \implies a = \text{Map.empty}$ 
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:  $a ++ b = \text{Map.empty} \implies b = \text{Map.empty}$ 
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!]:  $\llbracket a ++ b = \text{Map.empty}; \llbracket a = \text{Map.empty}; b = \text{Map.empty} \rrbracket \implies R \rrbracket \implies R$ 
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*]:
 $object\text{-}type\ (add\text{-}to\text{-}slots\ slots\ x) = object\text{-}type\ x$
by (*clarsimp simp: object-type-def add-to-slots-def update-slots-def split: cdl-object.splits*)

lemma *object-slots-update-slots* [*simp*]:
 $has\text{-}slots\ obj \implies object\text{-}slots\ (update\text{-}slots\ slots\ obj) = slots$
by (*clarsimp simp: object-slots-def update-slots-def has-slots-def split: cdl-object.splits*)

lemma *object-slots-update-slots-empty* [*simp*]:
 $\neg has\text{-}slots\ obj \implies object\text{-}slots\ (update\text{-}slots\ slots\ obj) = Map.empty$
by (*clarsimp simp: object-slots-def update-slots-def has-slots-def split: cdl-object.splits*)

lemma *update-slots-no-slots* [*simp*]:
 $\neg has\text{-}slots\ obj \implies update\text{-}slots\ slots\ obj = obj$
by (*clarsimp simp: update-slots-def has-slots-def split: cdl-object.splits*)

lemma *update-slots-update-slots* [*simp*]:
 $update\text{-}slots\ slots\ (update\text{-}slots\ slots'\ obj) = update\text{-}slots\ slots\ obj$
by (*clarsimp simp: update-slots-def split: cdl-object.splits*)

lemma *update-slots-same-object*:
 $a = b \implies update\text{-}slots\ a\ obj = update\text{-}slots\ b\ obj$
by (*erule arg-cong*)

lemma *object-type-has-slots*:
 $\llbracket has\text{-}slots\ x; object\text{-}type\ x = object\text{-}type\ y \rrbracket \implies has\text{-}slots\ y$
by (*clarsimp simp: object-type-def has-slots-def split: cdl-object.splits*)

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

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

lemma *object-slots-object-clean* [*simp*]:
 $object\text{-}slots\ (object\text{-}clean\ obj\ cmp) = clean\text{-}slots\ (object\text{-}slots\ obj)\ cmp$
by (*clarsimp simp: object-clean-def*)

lemma *object-slots-add-to-slots* [*simp*]:
 $object\text{-}type\ y = object\text{-}type\ z \implies object\text{-}slots\ (add\text{-}to\text{-}slots\ (object\text{-}slots\ y)\ z) = object\text{-}slots\ y ++ object\text{-}slots\ z$
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*]:

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

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

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

$\llbracket \text{None} \notin \text{cmps}; \text{None} \notin \text{cmps}'; \text{object-type } \text{obj} = \text{object-type } \text{obj}' \rrbracket$

$\implies \text{update-slots slots } (\text{object-clean-fields } \text{obj } \text{cmps}) =$
 $\text{update-slots slots } (\text{object-clean-fields } \text{obj}' \text{ cmps}')$

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

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

$\llbracket \text{None} \notin \text{cmps}; \text{None} \notin \text{cmps}'; \text{object-type } \text{obj} = \text{object-type } \text{obj}'; \neg \text{has-slots } \text{obj};$
 $\neg \text{has-slots } \text{obj}' \rrbracket$

$\implies \text{object-clean-fields } \text{obj } \text{cmps} = \text{object-clean-fields } \text{obj}' \text{ cmps}'$

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

lemma *update-slots-object-clean*:

$\llbracket \text{None} \notin \text{cmps}; \text{None} \notin \text{cmps}'; \text{object-type } \text{obj} = \text{object-type } \text{obj}' \rrbracket$

$\implies \text{update-slots slots } (\text{object-clean } \text{obj } \text{cmps}) = \text{update-slots slots } (\text{object-clean}$
 $\text{obj}' \text{ cmps}')$

apply (*clarsimp simp: object-clean-def object-clean-slots-def*)

apply (*erule (2) update-slots-object-clean-fields*)

done

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

$\forall \text{obj-id. not-conflicting-objects } x \ z \ \text{obj-id} \wedge$

$\text{not-conflicting-objects } y \ z \ \text{obj-id} \wedge$

$\text{not-conflicting-objects } x \ z \ \text{obj-id} \implies$

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

$\text{cdl-heap-add } x \ (\text{SepState } (\text{cdl-heap-add } y \ z) \ (\text{cdl-ghost-state-add } y \ z))$

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  $\neg$  has-slots obj-y)
apply (subgoal-tac  $\neg$  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]]
 $\implies$  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*:

```

cmpr-a  $\cap$  cmpr-b = {}
 $\implies$  clean-slots slots-a cmpr-a ++ clean-slots slots-b cmpr-b =
      clean-slots slots-b cmpr-b ++ clean-slots slots-a cmpr-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  $\implies$  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 cmpr'1={}) and obj'1=obj-a in trans [OF update-slots-object-clean-fields],
fastforce+)
done

```

lemma *object-add-comm*:
 $\llbracket \text{object-type } obj\text{-}a = \text{object-type } obj\text{-}b; \text{cmps}\text{-}a \cap \text{cmps}\text{-}b = \{\} \rrbracket$
 $\implies \text{object-add } obj\text{-}a \text{ } obj\text{-}b \text{ } \text{cmps}\text{-}a \text{ } \text{cmps}\text{-}b = \text{object-add } obj\text{-}b \text{ } obj\text{-}a \text{ } \text{cmps}\text{-}b \text{ } \text{cmps}\text{-}a$
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*:
 $\text{sep-state-disj } x \ y \implies \text{sep-state-add } x \ y = \text{sep-state-add } y \ x$
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*:
 $\llbracket \text{object-slots } y\text{-}obj \perp \text{object-slots } z\text{-}obj; \text{update-slots } \text{Map.empty } y\text{-}obj = \text{update-slots } \text{Map.empty } z\text{-}obj \rrbracket$
 $\implies \text{add-to-slots } (\text{object-slots } z\text{-}obj) \ y\text{-}obj = \text{add-to-slots } (\text{object-slots } y\text{-}obj) \ z\text{-}obj$
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 \implies (*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 \implies (*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'
 \implies *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'
 \implies *object-type (object-add obj obj' cmp cmp')* = *object-type obj'*
by (*clarsimp simp: object-add-def Let-unfold*)

lemma *sep-state-add-disjL*:

$\llbracket \text{sep-state-disj } y \ z; \text{ sep-state-disj } x \ (\text{sep-state-add } y \ z) \rrbracket \implies \text{sep-state-disj } x \ y$
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*:

$\llbracket \text{sep-state-disj } y \ z; \text{ sep-state-disj } x \ (\text{sep-state-add } y \ z) \rrbracket \implies \text{sep-state-disj } x \ z$
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*:

$\llbracket \text{sep-state-disj } y \ z; \text{ sep-state-disj } x \ y; \text{ sep-state-disj } x \ z \rrbracket \implies \text{sep-state-disj } x$
(*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 (λobj-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  $\Rightarrow$  bool

definition
  state-sep-projection :: cdl-state  $\Rightarrow$  sep-state
where
  state-sep-projection  $\equiv$   $\lambda s$ . SepState (cdl-objects s) (cdl-ghost-state s)

abbreviation
  lift' :: (sep-state  $\Rightarrow$  'a)  $\Rightarrow$  cdl-state  $\Rightarrow$  'a (<->)
where
  <P> s  $\equiv$  P (state-sep-projection s)

definition
  sep-map-general :: cdl-object-id  $\Rightarrow$  cdl-object  $\Rightarrow$  cdl-components  $\Rightarrow$  sep-pred
where
  sep-map-general p obj gs  $\equiv$   $\lambda s$ . sep-heap s = [p  $\mapsto$  obj]  $\wedge$  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.