

Algebra of Monotonic Boolean Transformers

Viorel Preoteasa

May 26, 2024

Abstract

Algebras of imperative programming languages have been successful in reasoning about programs. In general an algebra of programs is an algebraic structure with programs as elements and with program compositions (sequential composition, choice, skip) as algebra operations. Various versions of these algebras were introduced to model partial correctness, total correctness, refinement, demonic choice, and other aspects. We formalize here an algebra which can be used to model total correctness, refinement, demonic and angelic choice. The basic model of this algebra are monotonic Boolean transformers (monotonic functions from a Boolean algebra to itself).

Contents

1	Introduction	1
2	Monotonic Boolean Transformers	2
3	Algebra of Monotonic Boolean Transformers	12
3.1	Assertions	18
3.2	Weakest precondition of true	26
3.3	Monotonic Boolean transformers algebra with post condition statement	28
3.4	Complete monotonic Boolean transformers algebra	29
4	Boolean Algebra of Assertions	33
5	Program statements, Hoare and refinement rules	37

1 Introduction

Abstract algebra is a useful tool in mathematics. Rather than working with specific models like natural numbers and algebra of truth values, one could reason in a more abstract setting and obtain results which are more general

and applicable in different models. Algebras of logics are very important tools in studying various aspects of logical systems. Algebras of programming theories have also a significant contribution to the simplification of reasoning about programs. Programs are elements of an algebra and program compositions and program constants (sequential composition, choice, iteration, skip, fail) are the operations of the algebra. These operations satisfy a number of relations which are used for reasoning about programs. Kleene algebra with tests (KAT) [10] is an extension of Kleene algebra and it is suitable for reasoning about programs in a partial correctness framework. Various versions of Kleene algebras have been introduced, ranging from Kleene algebra with domain [7] and concurrent Kleene algebra [9] to an algebra for separation logic [6].

Refinement Calculus [1, 2, 5, 12] is a calculus based on (monotonic) predicate transformers suitable for program development in a total correctness framework. Within this calculus various aspects of imperative programming languages can be formalized. These include total correctness, partial correctness, demonic choice, and angelic choice. Demonic refinement algebra (DRA) was introduced in [15, 16] as a variation of KAT to allow also reasoning about total correctness. The intended model of DRA is the set of conjunctive predicate transformers and this algebra cannot represent angelic choice. General refinement algebra (GRA) was also introduced in [16], but few results were proved and they were mostly related to iteration. Although the intended model for GRA is the set of monotonic predicate transformers, GRA does not include the angelic choice operator. GRA has been further extended in [14] with enabledness and termination operators, and it was extended for probabilistic programs in [11].

This formalization is based on [13] where a different extension of GRA is introduced. In [13] GRA is extended with a dual operator [8, 3, 4, 5]. The intended model for this algebra is the set of monotonic Boolean transformers (monotonic functions from a Boolean algebra to itself).

This formalization is structured as follows. Section 2 introduces the monotonic Boolean transformers that are the basic model of the algebra. Section 3 introduces the monotonic Boolean transformers algebra and some of its properties. Section 4 introduces the Boolean algebra of assertions. In section 5 we introduce standard program statements and we prove their Hoare total correctness rules.

2 Monotonic Boolean Transformers

theory *Mono-Bool-Tran*

imports

LatticeProperties.Complete-Lattice-Prop

LatticeProperties.Conj-Disj

begin

The type of monotonic transformers is the type associated to the set of monotonic functions from a partially ordered set (poset) to itself. The type of monotonic transformers with the pointwise extended order is also a poset. The monotonic transformers with composition and identity form a monoid, and the monoid operation is compatible with the order.

Gradually we extend the algebraic structure of monotonic transformers to lattices, and complete lattices. We also introduce a dual operator $((\text{dual } f)p = -f(-p))$ on monotonic transformers over a boolean algebra. However the monotonic transformers over a boolean algebra are not closed to the pointwise extended negation operator.

Finally we introduce an iteration operator on monotonic transformers over a complete lattice.

unbundle *lattice-syntax*

lemma *Inf-comp-fun*:

$\prod M \circ f = (\prod m \in M. m \circ f)$
by (*simp add: fun-eq-iff image-comp*)

lemma *INF-comp-fun*:

$(\prod a \in A. g a) \circ f = (\prod a \in A. g a \circ f)$
by (*simp add: fun-eq-iff image-comp*)

lemma *Sup-comp-fun*:

$\bigsqcup M \circ f = (\bigsqcup m \in M. m \circ f)$
by (*simp add: fun-eq-iff image-comp*)

lemma *SUP-comp-fun*:

$(\bigsqcup a \in A. g a) \circ f = (\bigsqcup a \in A. g a \circ f)$
by (*simp add: fun-eq-iff image-comp*)

lemma (**in order**) *mono-const* [*simp*]:

mono $(\lambda-. c)$
by (*auto intro: monoI*)

lemma (**in order**) *mono-id* [*simp*]:

mono id
by (*auto intro: order-class.monoI*)

lemma (**in order**) *mono-comp* [*simp*]:

mono f \implies *mono g* \implies *mono (f o g)*
by (*auto intro!: monoI elim!: monoE order-class.monoE*)

lemma (**in bot**) *mono-bot* [*simp*]:

mono \perp
by (*auto intro: monoI*)

lemma (**in top**) *mono-top* [*simp*]:

```

mono  $\top$ 
by (auto intro: monoI)

lemma (in semilattice-inf) mono-inf [simp]:
  assumes mono f and mono g
  shows mono (f  $\sqcap$  g)
proof
  fix a b
  assume a  $\leq$  b
  have f a  $\sqcap$  g a  $\leq$  f a by simp
  also from  $\langle$ mono f $\rangle$   $\langle$ a  $\leq$  b $\rangle$  have ...  $\leq$  f b by (auto elim: monoE)
  finally have *: f a  $\sqcap$  g a  $\leq$  f b .
  have f a  $\sqcap$  g a  $\leq$  g a by simp
  also from  $\langle$ mono g $\rangle$   $\langle$ a  $\leq$  b $\rangle$  have ...  $\leq$  g b by (auto elim: monoE)
  finally have **: f a  $\sqcap$  g a  $\leq$  g b .
  from * ** show (f  $\sqcap$  g) a  $\leq$  (f  $\sqcap$  g) b by auto
qed

lemma (in semilattice-sup) mono-sup [simp]:
  assumes mono f and mono g
  shows mono (f  $\sqcup$  g)
proof
  fix a b
  assume a  $\leq$  b
  from  $\langle$ mono f $\rangle$   $\langle$ a  $\leq$  b $\rangle$  have f a  $\leq$  f b by (auto elim: monoE)
  also have f b  $\leq$  f b  $\sqcup$  g b by simp
  finally have *: f a  $\leq$  f b  $\sqcup$  g b .
  from  $\langle$ mono g $\rangle$   $\langle$ a  $\leq$  b $\rangle$  have g a  $\leq$  g b by (auto elim: monoE)
  also have g b  $\leq$  f b  $\sqcup$  g b by simp
  finally have **: g a  $\leq$  f b  $\sqcup$  g b .
  from * ** show (f  $\sqcup$  g) a  $\leq$  (f  $\sqcup$  g) b by auto
qed

lemma (in complete-lattice) mono-Inf [simp]:
  assumes A  $\subseteq$  {f :: 'a  $\Rightarrow$  'b:: complete-lattice. mono f}
  shows mono ( $\sqcap$  A)
proof
  fix a b
  assume a  $\leq$  b
  { fix f
    assume f  $\in$  A
    with assms have mono f by auto
    with  $\langle$ a  $\leq$  b $\rangle$  have f a  $\leq$  f b by (auto elim: monoE)
  }
  then have ( $\sqcap$  f $\in$ A. f a)  $\leq$  ( $\sqcap$  f $\in$ A. f b)
  by (auto intro: complete-lattice-class.INF-greatest complete-lattice-class.INF-lower2)
  then show ( $\sqcap$  A) a  $\leq$  ( $\sqcap$  A) b by simp
qed

```

```

lemma (in complete-lattice) mono-Sup [simp]:
  assumes  $A \subseteq \{f :: 'a \Rightarrow 'b :: \text{complete-lattice. mono } f\}$ 
  shows mono ( $\bigsqcup A$ )
proof
  fix  $a\ b$ 
  assume  $a \leq b$ 
  { fix  $f$ 
    assume  $f \in A$ 
    with assms have mono f by auto
    with  $\langle a \leq b \rangle$  have  $f\ a \leq f\ b$  by (auto elim: monoE)
  }
  then have  $(\bigsqcup f \in A. f\ a) \leq (\bigsqcup f \in A. f\ b)$ 
  by (auto intro: complete-lattice-class.SUP-least complete-lattice-class.SUP-upper2)
  then show  $(\bigsqcup A)\ a \leq (\bigsqcup A)\ b$  by simp
qed

typedef (overloaded) 'a MonoTran =  $\{f :: 'a :: \text{order} \Rightarrow 'a . \text{mono } f\}$ 
proof
  show  $id \in ?\text{MonoTran}$  by simp
qed

lemma [simp]:
  mono (Rep-MonoTran  $f$ )
  using Rep-MonoTran [of f] by simp

setup-lifting type-definition-MonoTran

instantiation MonoTran :: (order) order
begin

lift-definition less-eq-MonoTran :: 'a MonoTran  $\Rightarrow$  'a MonoTran  $\Rightarrow$  bool
  is less-eq .

lift-definition less-MonoTran :: 'a MonoTran  $\Rightarrow$  'a MonoTran  $\Rightarrow$  bool
  is less .

instance
  by intro-classes (transfer, auto intro: order-antisym)+

end

instantiation MonoTran :: (order) monoid-mult
begin

lift-definition one-MonoTran :: 'a MonoTran
  is id
  by (fact mono-id)

lift-definition times-MonoTran :: 'a MonoTran  $\Rightarrow$  'a MonoTran  $\Rightarrow$  'a MonoTran

```

```

is comp
by (fact mono-comp)

instance
by intro-classes (transfer, auto)+

end

instantiation MonoTran :: (order-bot) order-bot
begin

lift-definition bot-MonoTran :: 'a MonoTran
is  $\perp$ 
by (fact mono-bot)

instance
by intro-classes (transfer, simp)

end

instantiation MonoTran :: (order-top) order-top
begin

lift-definition top-MonoTran :: 'a MonoTran
is  $\top$ 
by (fact mono-top)

instance
by intro-classes (transfer, simp)

end

instantiation MonoTran :: (lattice) lattice
begin

lift-definition inf-MonoTran :: 'a MonoTran  $\Rightarrow$  'a MonoTran  $\Rightarrow$  'a MonoTran
is inf
by (fact mono-inf)

lift-definition sup-MonoTran :: 'a MonoTran  $\Rightarrow$  'a MonoTran  $\Rightarrow$  'a MonoTran
is sup
by (fact mono-sup)

instance
by intro-classes (transfer, simp)+

end

instance MonoTran :: (distrib-lattice) distrib-lattice

```

```

    by intro-classes (transfer, rule sup-inf-distrib1)

instantiation MonoTran :: (complete-lattice) complete-lattice
begin

lift-definition Inf-MonoTran :: 'a MonoTran set  $\Rightarrow$  'a MonoTran
  is Inf
  by (rule mono-Inf) auto

lift-definition Sup-MonoTran :: 'a MonoTran set  $\Rightarrow$  'a MonoTran
  is Sup
  by (rule mono-Sup) auto

instance
  by intro-classes (transfer, simp add: Inf-lower Sup-upper Inf-greatest Sup-least)+

end

context includes lifting-syntax
begin

lemma [transfer-rule]:
  (rel-set A  $\implies$  (A  $\implies$  pcr-MonoTran HOL.eq)  $\implies$  pcr-MonoTran
  HOL.eq) ( $\lambda A f. \sqcap (f \text{ ' } A)$ ) ( $\lambda A f. \sqcap (f \text{ ' } A)$ )
  by transfer-prover

lemma [transfer-rule]:
  (rel-set A  $\implies$  (A  $\implies$  pcr-MonoTran HOL.eq)  $\implies$  pcr-MonoTran
  HOL.eq) ( $\lambda A f. \sqcup (f \text{ ' } A)$ ) ( $\lambda A f. \sqcup (f \text{ ' } A)$ )
  by transfer-prover

end

instance MonoTran :: (complete-distrib-lattice) complete-distrib-lattice
proof (intro-classes, transfer)
  fix A :: ('a  $\Rightarrow$  'a) set set
  assume  $\forall A \in A. \text{Ball } A \text{ mono}$ 
  from this have [simp]:  $\{f \text{ ' } A \mid f. \forall Y \in A. f Y \in Y\} = \{x. (\exists f. (\forall x. (\forall x \in x. \text{mono } x) \longrightarrow \text{mono } (f x)) \wedge x = f \text{ ' } A \wedge (\forall Y \in A. f Y \in Y)) \wedge (\forall x \in x. \text{mono } x)\}$ 
  apply safe
  apply (rule-tac  $x = \lambda x. \text{if } x \in A \text{ then } f x \text{ else } \perp$  in exI)
  apply (simp add: if-split image-def)
  by blast+

  show  $\sqcap (Sup \text{ ' } A) \leq \sqcup (Inf \text{ ' } \{x. (\exists f \in \text{Collect } (\text{pred-fun } (\lambda A. \text{Ball } A \text{ mono})) \text{ mono}). x = f \text{ ' } A \wedge (\forall Y \in A. f Y \in Y)) \wedge \text{Ball } x \text{ mono}\}$ 
  by (simp add: Inf-Sup)
qed

```

definition

$dual\text{-}fun (f :: 'a :: boolean\text{-}algebra \Rightarrow 'a) = uminus \circ f \circ uminus$

lemma *dual-fun-apply* [simp]:

$dual\text{-}fun f p = - f (- p)$
by (*simp add: dual-fun-def*)

lemma *mono-dual-fun* [simp]:

$mono f \Longrightarrow mono (dual\text{-}fun f)$
apply (*rule monoI*)
apply (*erule monoE*)
apply *auto*
done

lemma (**in** *order*) *mono-inf-fun* [simp]:

fixes $x :: 'b :: semilattice\text{-}inf$
shows $mono (inf x)$
by (*auto intro!: order-class.monoI semilattice-inf-class.inf-mono*)

lemma (**in** *order*) *mono-sup-fun* [simp]:

fixes $x :: 'b :: semilattice\text{-}sup$
shows $mono (sup x)$
by (*auto intro!: order-class.monoI semilattice-sup-class.sup-mono*)

lemma *mono-comp-fun*:

fixes $f :: 'a :: order \Rightarrow 'b :: order$
shows $mono f \Longrightarrow mono ((\circ) f)$
by (*rule monoI*) (*auto simp add: le-fun-def elim: monoE*)

definition

$\Omega\text{-}fun f g = inf g \circ comp f$

lemma *Omega-fun-apply* [simp]:

$\Omega\text{-}fun f g h p = (g p \sqcap f (h p))$
by (*simp add: Omega-fun-def*)

lemma *mono-Omega-fun* [simp]:

$mono f \Longrightarrow mono (\Omega\text{-}fun f g)$
unfolding *Omega-fun-def*
by (*auto intro: mono-comp mono-comp-fun*)

lemma *mono-mono-Omega-fun* [simp]:

fixes $f :: 'b :: order \Rightarrow 'a :: semilattice\text{-}inf$ **and** $g :: 'c :: semilattice\text{-}inf \Rightarrow 'a$
shows $mono f \Longrightarrow mono g \Longrightarrow mono\text{-}mono (\Omega\text{-}fun f g)$
apply (*auto simp add: mono-mono-def Omega-fun-def*)
apply (*rule mono-comp*)
apply (*rule mono-inf-fun*)


```

apply (rule mono-comp-fun)
apply assumption
done

```

definition

```

omega-fun f = lfp (Omega-fun f id)

```

definition

```

star-fun f = gfp (Omega-fun f id)

```

lemma mono-omega-fun [simp]:

```

fixes f :: 'a::complete-lattice ⇒ 'a
assumes mono f
shows mono (omega-fun f)

```

proof

```

fix a b :: 'a
assume a ≤ b
from assms have mono (lfp (Omega-fun f id))
  by (auto intro: mono-mono-Omega-fun)
with ⟨a ≤ b⟩ show omega-fun f a ≤ omega-fun f b
  by (auto simp add: omega-fun-def elim: monoE)

```

qed

lemma mono-star-fun [simp]:

```

fixes f :: 'a::complete-lattice ⇒ 'a
assumes mono f
shows mono (star-fun f)

```

proof

```

fix a b :: 'a
assume a ≤ b
from assms have mono (gfp (Omega-fun f id))
  by (auto intro: mono-mono-Omega-fun)
with ⟨a ≤ b⟩ show star-fun f a ≤ star-fun f b
  by (auto simp add: star-fun-def elim: monoE)

```

qed

lemma lfp-omega-lowerbound:

```

mono f ⇒ Omega-fun f g A ≤ A ⇒ omega-fun f ∘ g ≤ A

```

```

apply (simp add: omega-fun-def)

```

```

apply (rule-tac P = λ x . x ∘ g ≤ A and f = Omega-fun f id in lfp-ordinal-induct)

```

```

apply simp-all

```

```

apply (simp add: le-fun-def o-def inf-fun-def id-def Omega-fun-def)

```

```

apply auto

```

```

apply (rule-tac y = f (A x) ⊓ g x in order-trans)

```

```

apply simp-all

```

```

apply (rule-tac y = f (S (g x)) in order-trans)

```

```

apply simp-all

```

```

apply (simp add: mono-def) apply (auto simp add: ac-simps)

```

```

apply (unfold Sup-comp-fun)

```

apply (*rule SUP-least*)
by *auto*

lemma *gfp-omega-upperbound:*

mono f $\implies A \leq \text{Omega-fun } f \ g \ A \implies A \leq \text{star-fun } f \circ g$

apply (*simp add: star-fun-def*)

apply (*rule-tac* $P = \lambda x . A \leq x \circ g$ **and** $f = \text{Omega-fun } f \ \text{id}$ **in** *gfp-ordinal-induct*)

apply *simp-all*

apply (*simp add: le-fun-def o-def inf-fun-def id-def Omega-fun-def*)

apply *auto*

apply (*rule-tac* $y = f \ (A \ x) \sqcap g \ x$ **in** *order-trans*)

apply *simp-all*

apply (*rule-tac* $y = f \ (A \ x)$ **in** *order-trans*)

apply *simp-all*

apply (*simp add: mono-def*)

apply (*unfold Inf-comp-fun*)

apply (*rule INF-greatest*)

by *auto*

lemma *lfp-omega-greatest:*

assumes $\bigwedge u. \text{Omega-fun } f \ g \ u \leq u \implies A \leq u$

shows $A \leq \text{omega-fun } f \circ g$

apply (*unfold omega-fun-def*)

apply (*simp add: lfp-def*)

apply (*unfold Inf-comp-fun*)

apply (*rule INF-greatest*)

apply *simp*

apply (*rule assms*)

apply (*simp add: le-fun-def*)

done

lemma *gfp-star-least:*

assumes $\bigwedge u. u \leq \text{Omega-fun } f \ g \ u \implies u \leq A$

shows $\text{star-fun } f \circ g \leq A$

apply (*unfold star-fun-def*)

apply (*simp add: gfp-def*)

apply (*unfold Sup-comp-fun*)

apply (*rule SUP-least*)

apply *simp*

apply (*rule assms*)

apply (*simp add: le-fun-def*)

done

lemma *lfp-omega:*

mono f $\implies \text{omega-fun } f \circ g = \text{lfp} \ (\text{Omega-fun } f \ g)$

apply (*rule antisym*)

apply (*rule lfp-omega-lowerbound*)

apply *simp-all*

apply (*simp add: lfp-def*)

```

apply (rule Inf-greatest)
apply safe
apply (rule-tac  $y = \text{Omega-fun } f \ g \ x$  in order-trans)
apply simp-all
apply (rule-tac  $f = \text{Omega-fun } f \ g$  in monoD)
apply simp-all
apply (rule Inf-lower)
apply simp
apply (rule lfp-omega-greatest)
apply (simp add: lfp-def)
apply (rule Inf-lower)
by simp

```

lemma *gfp-star*:

```

 $\text{mono } f \implies \text{star-fun } f \circ g = \text{gfp } (\text{Omega-fun } f \ g)$ 
apply (rule antisym)
apply (rule gfp-star-least)
apply (simp add: gfp-def)
apply (rule Sup-upper, simp)
apply (rule gfp-omega-upperbound)
apply simp-all
apply (simp add: gfp-def)
apply (rule Sup-least)
apply safe
apply (rule-tac  $y = \text{Omega-fun } f \ g \ x$  in order-trans)
apply simp-all
apply (rule-tac  $f = \text{Omega-fun } f \ g$  in monoD)
apply simp-all
apply (rule Sup-upper)
by simp

```

definition

```

 $\text{assert-fun } p \ q = (p \sqcap q :: 'a::\text{semilattice-inf})$ 

```

lemma *mono-assert-fun* [*simp*]:

```

 $\text{mono } (\text{assert-fun } p)$ 
apply (simp add: assert-fun-def mono-def, safe)
by (rule-tac  $y = x$  in order-trans, simp-all)

```

lemma *assert-fun-le-id* [*simp*]: $\text{assert-fun } p \leq \text{id}$

```

by (simp add: assert-fun-def id-def le-fun-def)

```

lemma *assert-fun-disjunctive* [*simp*]: $\text{assert-fun } (p::'a::\text{distrib-lattice}) \in \text{Apply.disjunctive}$

```

by (simp add: assert-fun-def Apply.disjunctive-def inf-sup-distrib)

```

definition

```

 $\text{assertion-fun} = \text{range } \text{assert-fun}$ 

```

lemma *assert-cont*:

```

(x :: 'a::boolean-algebra ⇒ 'a) ≤ id ⇒ x ∈ Apply.disjunctive ⇒ x = assert-fun
(x ⊔)
  apply (rule antisym)
  apply (simp-all add: le-fun-def assert-fun-def, safe)
  apply (rule-tac f = x in monoD, simp-all)
  apply (subgoal-tac x top = sup (x xa) (x (-xa)))
  apply simp
  apply (subst inf-sup-distrib)
  apply simp
  apply (rule-tac y = inf (- xa) xa in order-trans)
  supply [[simproc del: boolean-algebra-cancel-inf]]
  apply (simp del: compl-inf-bot)
  apply (rule-tac y = x (- xa) in order-trans)
  apply simp
  apply simp
  apply simp
  apply (cut-tac x = x and y = xa and z = -xa in Apply.disjunctiveD, simp)
  apply (subst (asm) sup-commute)
  apply (subst (asm) compl-sup-top)
  by simp

```

```

lemma assertion-fun-disj-less-one: assertion-fun = Apply.disjunctive ∩ {x::'a::boolean-algebra
⇒ 'a . x ≤ id}
  apply safe
  apply (simp-all add: assertion-fun-def, auto simp add: image-def)
  apply (rule-tac x = x ⊔ in exI)
  by (rule assert-cont, simp-all)

```

```

lemma assert-fun-dual: ((assert-fun p) o ⊔) ∩ (dual-fun (assert-fun p)) = as-
sert-fun p
  by (simp add: fun-eq-iff inf-fun-def dual-fun-def o-def assert-fun-def top-fun-def
inf-sup-distrib)

```

```

lemma assertion-fun-dual: x ∈ assertion-fun ⇒ (x o ⊔) ∩ (dual-fun x) = x
  by (simp add: assertion-fun-def, safe, simp add: assert-fun-dual)

```

```

lemma assertion-fun-MonoTran [simp]: x ∈ assertion-fun ⇒ mono x
  by (unfold assertion-fun-def, auto)

```

```

lemma assertion-fun-le-one [simp]: x ∈ assertion-fun ⇒ x ≤ id
  by (unfold assertion-fun-def, auto)

```

end

3 Algebra of Monotonic Boolean Transformers

```

theory Mono-Bool-Tran-Algebra
imports Mono-Bool-Tran
begin

```

In this section we introduce the *algebra of monotonic boolean transformers*. This is a bounded distributive lattice with a monoid operation, a dual operator and an iteration operator. The standard model for this algebra is the set of monotonic boolean transformers introduced in the previous section.

```

class dual =
  fixes dual::'a ⇒ 'a (- ^ o [81] 80)

class omega =
  fixes omega::'a ⇒ 'a (- ^ ω [81] 80)

class star =
  fixes star::'a ⇒ 'a ((- ^ *) [81] 80)

class dual-star =
  fixes dual-star::'a ⇒ 'a ((- ^ ⊗) [81] 80)

class mbt-algebra = monoid-mult + dual + omega + distrib-lattice + order-top +
order-bot + star + dual-star +
  assumes
    dual-le: (x ≤ y) = (y ^ o ≤ x ^ o)
  and dual-dual [simp]: (x ^ o) ^ o = x
  and dual-comp: (x * y) ^ o = x ^ o * y ^ o
  and dual-one [simp]: 1 ^ o = 1
  and top-comp [simp]: ⊤ * x = ⊤
  and inf-comp: (x ⊓ y) * z = (x * z) ⊓ (y * z)
  and le-comp: x ≤ y ⇒ z * x ≤ z * y
  and dual-neg: (x * ⊤) ⊓ (x ^ o * ⊥) = ⊥
  and omega-fix: x ^ ω = (x * (x ^ ω)) ⊓ 1
  and omega-least: (x * z) ⊓ y ≤ z ⇒ (x ^ ω) * y ≤ z
  and star-fix: x ^ * = (x * (x ^ *)) ⊓ 1
  and star-greatest: z ≤ (x * z) ⊓ y ⇒ z ≤ (x ^ *) * y
  and dual-star-def: (x ^ ⊗) = (((x ^ o) ^ *) ^ o)
begin

lemma le-comp-right: x ≤ y ⇒ x * z ≤ y * z
  apply (cut-tac x = x and y = y and z = z in inf-comp)
  apply (simp add: inf-absorb1)
  apply (subgoal-tac x * z ⊓ (y * z) ≤ y * z)
  apply simp
  by (rule inf-le2)

subclass bounded-lattice
  proof qed

end

instantiation MonoTran :: (complete-boolean-algebra) mbt-algebra
begin

```

lift-definition *dual-MonoTran* :: 'a MonoTran \Rightarrow 'a MonoTran
 is *dual-fun*
 by (fact mono-dual-fun)

lift-definition *omega-MonoTran* :: 'a MonoTran \Rightarrow 'a MonoTran
 is *omega-fun*
 by (fact mono-omega-fun)

lift-definition *star-MonoTran* :: 'a MonoTran \Rightarrow 'a MonoTran
 is *star-fun*
 by (fact mono-star-fun)

definition *dual-star-MonoTran* :: 'a MonoTran \Rightarrow 'a MonoTran
 where
 $(x::('a MonoTran)) \hat{\otimes} = ((x \hat{o}) \hat{*}) \hat{o}$

instance proof

fix $x\ y :: 'a MonoTran$ **show** $(x \leq y) = (y \hat{o} \leq x \hat{o})$
 apply *transfer*
 apply (auto simp add: fun-eq-iff le-fun-def)
 apply (drule-tac $x = -xa$ in spec)
 apply *simp*
 done

next

fix $x :: 'a MonoTran$ **show** $(x \hat{o}) \hat{o} = x$
 apply *transfer*
 apply (simp add: fun-eq-iff)
 done

next

fix $x\ y :: 'a MonoTran$ **show** $(x * y) \hat{o} = x \hat{o} * y \hat{o}$
 apply *transfer*
 apply (simp add: fun-eq-iff)
 done

next

show $(1::'a MonoTran) \hat{o} = 1$
 apply *transfer*
 apply (simp add: fun-eq-iff)
 done

next

fix $x :: 'a MonoTran$ **show** $\top * x = \top$
 apply *transfer*
 apply (simp add: fun-eq-iff)
 done

next

fix $x\ y\ z :: 'a MonoTran$ **show** $(x \sqcap y) * z = (x * z) \sqcap (y * z)$
 apply *transfer*
 apply (simp add: fun-eq-iff)
 done

next

```

fix x y z :: 'a MonoTran assume A: x ≤ y from A show z * x ≤ z * y
  apply transfer
  apply (auto simp add: le-fun-def elim: monoE)
  done
next
fix x :: 'a MonoTran show x * ⊤ ⊓ (x ^ o * ⊥) = ⊥
  apply transfer
  apply (simp add: fun-eq-iff)
  done
next
fix x :: 'a MonoTran show x ^ ω = x * x ^ ω ⊓ 1
  apply transfer
  apply (simp add: fun-eq-iff)
  apply (simp add: omega-fun-def Omega-fun-def)
  apply (subst lfp-unfold, simp-all add: ac-simps)
  apply (auto intro!: mono-comp mono-comp-fun)
  done
next
fix x y z :: 'a MonoTran assume A: x * z ⊓ y ≤ z from A show x ^ ω * y ≤ z
  apply transfer
  apply (auto simp add: lfp-omega lfp-def)
  apply (rule Inf-lower)
  apply (auto simp add: Omega-fun-def ac-simps)
  done
next
fix x :: 'a MonoTran show x ^ * = x * x ^ * ⊓ 1
  apply transfer
  apply (auto simp add: star-fun-def Omega-fun-def)
  apply (subst gfp-unfold, simp-all add: ac-simps)
  apply (auto intro!: mono-comp mono-comp-fun)
  done
next
fix x y z :: 'a MonoTran assume A: z ≤ x * z ⊓ y from A show z ≤ x ^ * * y
  apply transfer
  apply (auto simp add: gfp-star gfp-def)
  apply (rule Sup-upper)
  apply (auto simp add: Omega-fun-def)
  done
next
fix x :: 'a MonoTran show x ^ ⊗ = ((x ^ o) ^ *) ^ o
  by (simp add: dual-star-MonoTran-def)
qed

end

context mbt-algebra begin

lemma dual-top [simp]: ⊤ ^ o = ⊥
  apply (rule order.antisym, simp-all)

```

by (*subst dual-le, simp*)

lemma *dual-bot* [*simp*]: $\perp \hat{o} = \top$
apply (*rule order.antisym, simp-all*)
by (*subst dual-le, simp*)

lemma *dual-inf*: $(x \sqcap y) \hat{o} = (x \hat{o}) \sqcup (y \hat{o})$
apply (*rule order.antisym, simp-all, safe*)
apply (*subst dual-le, simp, safe*)
apply (*subst dual-le, simp*)
apply (*subst dual-le, simp*)
apply (*subst dual-le, simp*)
by (*subst dual-le, simp*)

lemma *dual-sup*: $(x \sqcup y) \hat{o} = (x \hat{o}) \sqcap (y \hat{o})$
apply (*rule order.antisym, simp-all, safe*)
apply (*subst dual-le, simp*)
apply (*subst dual-le, simp*)
apply (*subst dual-le, simp, safe*)
apply (*subst dual-le, simp*)
by (*subst dual-le, simp*)

lemma *sup-comp*: $(x \sqcup y) * z = (x * z) \sqcup (y * z)$
apply (*subgoal-tac ((x \hat{o} \sqcap y \hat{o}) * z \hat{o}) \hat{o} = ((x \hat{o} * z \hat{o}) \sqcap (y \hat{o} * z \hat{o})) \hat{o}*)
apply (*simp add: dual-inf dual-comp*)
by (*simp add: inf-comp*)

lemma *dual-eq*: $x \hat{o} = y \hat{o} \implies x = y$
apply (*subgoal-tac (x \hat{o}) \hat{o} = (y \hat{o}) \hat{o}*)
apply (*subst (asm) dual-dual*)
apply (*subst (asm) dual-dual*)
by *simp-all*

lemma *dual-neg-top* [*simp*]: $(x \hat{o} * \perp) \sqcup (x * \top) = \top$
apply (*rule dual-eq*)
by(*simp add: dual-sup dual-comp dual-neg*)

lemma *bot-comp* [*simp*]: $\perp * x = \perp$
by (*rule dual-eq, simp add: dual-comp*)

lemma [*simp*]: $(x * \top) * y = x * \top$
by (*simp add: mult.assoc*)

lemma [*simp*]: $(x * \perp) * y = x * \perp$
by (*simp add: mult.assoc*)

lemma *gt-one-comp*: $1 \leq x \implies y \leq x * y$
by (*cut-tac* $x = 1$ **and** $y = x$ **and** $z = y$ **in** *le-comp-right, simp-all*)

theorem *omega-comp-fix*: $x \hat{\omega} * y = (x * (x \hat{\omega})) * y \sqcap y$
apply (*subst omega-fix*)
by (*simp add: inf-comp*)

theorem *dual-star-fix*: $x \hat{\otimes} = (x * (x \hat{\otimes})) \sqcup 1$
by (*metis dual-comp dual-dual dual-inf dual-one dual-star-def star-fix*)

theorem *star-comp-fix*: $x \hat{*} * y = (x * (x \hat{*})) * y \sqcap y$
apply (*subst star-fix*)
by (*simp add: inf-comp*)

theorem *dual-star-comp-fix*: $x \hat{\otimes} * y = (x * (x \hat{\otimes})) * y \sqcup y$
apply (*subst dual-star-fix*)
by (*simp add: sup-comp*)

theorem *dual-star-least*: $(x * z) \sqcup y \leq z \implies (x \hat{\otimes}) * y \leq z$
apply (*subst dual-le*)
apply (*simp add: dual-star-def dual-comp*)
apply (*rule star-greatest*)
apply (*subst dual-le*)
by (*simp add: dual-inf dual-comp*)

lemma *omega-one* [*simp*]: $1 \hat{\omega} = \perp$
apply (*rule order.antisym, simp-all*)
by (*cut-tac* $x = 1::'a$ **and** $y = 1$ **and** $z = \perp$ **in** *omega-least, simp-all*)

lemma *omega-mono*: $x \leq y \implies x \hat{\omega} \leq y \hat{\omega}$
apply (*cut-tac* $x = x$ **and** $y = 1$ **and** $z = y \hat{\omega}$ **in** *omega-least, simp-all*)
apply (*subst* (2) *omega-fix, simp-all*)
apply (*rule-tac* $y = x * y \hat{\omega}$ **in** *order-trans, simp*)
by (*rule le-comp-right, simp*)

end

sublocale *mbt-algebra* < *conjunctive inf inf times*
done

sublocale *mbt-algebra* < *disjunctive sup sup times*
done

context *mbt-algebra* **begin**

lemma *dual-conjunctive*: $x \in \text{conjunctive} \implies x \hat{o} \in \text{disjunctive}$
apply (*simp add: conjunctive-def disjunctive-def*)
apply *safe*
apply (*rule dual-eq*)
by (*simp add: dual-comp dual-sup*)

lemma *dual-disjunctive*: $x \in \text{disjunctive} \implies x \hat{\ } o \in \text{conjunctive}$
apply (*simp add: conjunctive-def disjunctive-def*)
apply *safe*
apply (*rule dual-eq*)
by (*simp add: dual-comp dual-inf*)

lemma *comp-pres-conj*: $x \in \text{conjunctive} \implies y \in \text{conjunctive} \implies x * y \in \text{conjunctive}$
apply (*subst conjunctive-def, safe*)
by (*simp add: mult.assoc conjunctiveD*)

lemma *comp-pres-disj*: $x \in \text{disjunctive} \implies y \in \text{disjunctive} \implies x * y \in \text{disjunctive}$
apply (*subst disjunctive-def, safe*)
by (*simp add: mult.assoc disjunctiveD*)

lemma *start-pres-conj*: $x \in \text{conjunctive} \implies (x \hat{\ } *) \in \text{conjunctive}$
apply (*subst conjunctive-def, safe*)
apply (*rule order.antisym, simp-all*)
apply (*metis inf-le1 inf-le2 le-comp*)
apply (*rule star-greatest*)
apply (*subst conjunctiveD, simp*)
apply (*subst star-comp-fix*)
apply (*subst star-comp-fix*)
by (*metis inf.assoc inf-left-commute mult.assoc order-refl*)

lemma *dual-star-pres-disj*: $x \in \text{disjunctive} \implies x \hat{\ } \otimes \in \text{disjunctive}$
apply (*simp add: dual-star-def*)
apply (*rule dual-conjunctive*)
apply (*rule start-pres-conj*)
by (*rule dual-disjunctive, simp*)

3.1 Assertions

Usually, in Kleene algebra with tests or in other program algebras, tests or assertions or assumptions are defined using an existential quantifier. An element of the algebra is a test if it has a complement with respect to \perp and 1 . In this formalization assertions can be defined much simpler using the dual operator.

definition

$$\text{assertion} = \{x \mid x \leq 1 \wedge (x * \top) \sqcap (x \hat{\ } o) = x\}$$

lemma *assertion-prop*: $x \in \text{assertion} \implies (x * \top) \sqcap 1 = x$
apply (*simp add: assertion-def*)
apply *safe*
apply (*rule order.antisym*)
apply *simp-all*
proof –
assume [*simp*]: $x \leq 1$

```

assume  $A: x * \top \sqcap x \hat{o} = x$ 
have  $x * \top \sqcap 1 \leq x * \top \sqcap x \hat{o}$ 
  apply simp
  apply (rule-tac  $y = 1$  in order-trans)
  apply simp
  apply (subst dual-le)
  by simp
also have  $\dots = x$  by (cut-tac  $A$ , simp)
finally show  $x * \top \sqcap 1 \leq x$  .
next
assume  $A: x * \top \sqcap x \hat{o} = x$ 
have  $x = x * \top \sqcap x \hat{o}$  by (simp add: A)
also have  $\dots \leq x * \top$  by simp
finally show  $x \leq x * \top$  .
qed

```

```

lemma dual-assertion-prop:  $x \in \text{assertion} \implies ((x \hat{o}) * \perp) \sqcup 1 = x \hat{o}$ 
apply (rule dual-eq)
by (simp add: dual-sup dual-comp assertion-prop)

```

```

lemma assertion-disjunctive:  $x \in \text{assertion} \implies x \in \text{disjunctive}$ 
apply (simp add: disjunctive-def, safe)
apply (drule assertion-prop)
proof –
  assume  $A: x * \top \sqcap 1 = x$ 
  fix  $y z :: 'a$ 
  have  $x * (y \sqcup z) = (x * \top \sqcap 1) * (y \sqcup z)$  by (cut-tac A, simp)
  also have  $\dots = (x * \top) \sqcap (y \sqcup z)$  by (simp add: inf-comp)
  also have  $\dots = ((x * \top) \sqcap y) \sqcup ((x * \top) \sqcap z)$  by (simp add: inf-sup-distrib)
  also have  $\dots = (((x * \top) \sqcap 1) * y) \sqcup (((x * \top) \sqcap 1) * z)$  by (simp add: inf-comp)
  also have  $\dots = x * y \sqcup x * z$  by (cut-tac A, simp)
  finally show  $x * (y \sqcup z) = x * y \sqcup x * z$  .
qed

```

```

lemma Abs-MonoTran-injective:  $\text{mono } x \implies \text{mono } y \implies \text{Abs-MonoTran } x = \text{Abs-MonoTran } y \implies x = y$ 
apply (subgoal-tac Rep-MonoTran (Abs-MonoTran x) = Rep-MonoTran (Abs-MonoTran y))
apply (subst (asm) Abs-MonoTran-inverse, simp)
by (subst (asm) Abs-MonoTran-inverse, simp-all)
end

```

```

lemma mbta-MonoTran-disjunctive:  $\text{Rep-MonoTran } \text{'disjunctive} = \text{Apply.disjunctive}$ 
apply (simp add: disjunctive-def Apply.disjunctive-def)
apply transfer
apply auto
proof –
  fix  $f :: 'a \Rightarrow 'a$  and  $a b$ 

```

```

assume prem:  $\forall y. \text{mono } y \longrightarrow (\forall z. \text{mono } z \longrightarrow f \circ y \sqcup z = (f \circ y) \sqcup (f \circ z))$ 
{ fix g h :: 'b  $\Rightarrow$  'a
  assume mono g and mono h
  then have  $f \circ g \sqcup h = (f \circ g) \sqcup (f \circ h)$ 
    using prem by blast
} note * = this
assume mono f
show  $f (a \sqcup b) = f a \sqcup f b$  (is ?P = ?Q)
proof (rule order-antisym)
  show ?P  $\leq$  ?Q
    using * [of  $\lambda-. a \lambda-. b$ ] by (simp add: comp-def fun-eq-iff)
next
  from  $\langle \text{mono } f \rangle$  show ?Q  $\leq$  ?P
    using Fun.semilattice-sup-class.mono-sup by blast
qed
next
fix f :: 'a  $\Rightarrow$  'a
assume  $\forall y z. f (y \sqcup z) = f y \sqcup f z$ 
then have *:  $\bigwedge y z. f (y \sqcup z) = f y \sqcup f z$  by blast
show mono f
proof
  fix a b :: 'a
  assume  $a \leq b$ 
  then show  $f a \leq f b$ 
    unfolding sup.order-iff * [symmetric] by simp
qed
qed

lemma assertion-MonoTran: assertion = Abs-MonoTran ' assertion-fun
  apply (safe)
  apply (subst assertion-fun-disj-less-one)
  apply (simp add: image-def)
  apply (rule-tac x = Rep-MonoTran x in beXI)
  apply (simp add: Rep-MonoTran-inverse)
  apply safe
  apply (drule assertion-disjunctive)
  apply (unfold mbta-MonoTran-disjunctive [THEN sym], simp)
  apply (simp add: assertion-def less-eq-MonoTran-def one-MonoTran-def Abs-MonoTran-inverse)
  apply (simp add: assertion-def)
  by (simp-all add: inf-MonoTran-def less-eq-MonoTran-def
    times-MonoTran-def dual-MonoTran-def top-MonoTran-def Abs-MonoTran-inverse
    one-MonoTran-def assertion-fun-dual)

context mbt-algebra begin
lemma assertion-conjunctive:  $x \in \text{assertion} \Longrightarrow x \in \text{conjunctive}$ 
  apply (simp add: conjunctive-def, safe)
  apply (drule assertion-prop)
proof –
  assume A:  $x * \top \sqcap 1 = x$ 

```

```

fix y z::'a
have x * (y  $\sqcap$  z) = (x *  $\top$   $\sqcap$  1) * (y  $\sqcap$  z) by (cut-tac A, simp)
also have ... = (x *  $\top$ )  $\sqcap$  (y  $\sqcap$  z) by (simp add: inf-comp)
also have ... = ((x *  $\top$ )  $\sqcap$  y)  $\sqcap$  ((x *  $\top$ )  $\sqcap$  z)
  apply (rule order.antisym, simp-all, safe)
  apply (rule-tac y = y  $\sqcap$  z in order-trans)
  apply (rule inf-le2)
  apply simp
  apply (rule-tac y = y  $\sqcap$  z in order-trans)
  apply (rule inf-le2)
  apply simp-all
  apply (simp add: inf-assoc)
  apply (rule-tac y = x *  $\top$   $\sqcap$  y in order-trans)
  apply (rule inf-le1)
  apply simp
  apply (rule-tac y = x *  $\top$   $\sqcap$  z in order-trans)
  apply (rule inf-le2)
  by simp
also have ... = (((x *  $\top$ )  $\sqcap$  1) * y)  $\sqcap$  (((x *  $\top$ )  $\sqcap$  1) * z) by (simp add:
inf-comp)
also have ... = (x * y)  $\sqcap$  (x * z) by (cut-tac A, simp)
finally show x * (y  $\sqcap$  z) = (x * y)  $\sqcap$  (x * z) .
qed

```

lemma dual-assertion-conjunctive: $x \in \text{assertion} \implies x \hat{o} \in \text{conjunctive}$
apply (drule assertion-disjunctive)
by (rule dual-disjunctive, simp)

lemma dual-assertion-disjunct: $x \in \text{assertion} \implies x \hat{o} \in \text{disjunctive}$
apply (drule assertion-conjunctive)
by (rule dual-conjunctive, simp)

lemma [simp]: $x \in \text{assertion} \implies y \in \text{assertion} \implies x \sqcap y \leq x * y$
apply (simp add: assertion-def, safe)
proof –
assume A: $x \leq 1$
assume B: $x * \top \sqcap x \hat{o} = x$
assume C: $y \leq 1$
assume D: $y * \top \sqcap y \hat{o} = y$
have $x \sqcap y = (x * \top \sqcap x \hat{o}) \sqcap (y * \top \sqcap y \hat{o})$ **by** (cut-tac B D, simp)
also have ... $\leq (x * \top) \sqcap (((x \hat{o}) * (y * \top)) \sqcap ((x \hat{o}) * (y \hat{o})))$
apply (simp, safe)
apply (rule-tac y = x * \top \sqcap x \hat{o} **in** order-trans)
apply (rule inf-le1)
apply simp
apply (rule-tac y = y * \top **in** order-trans)
apply (rule-tac y = y * \top \sqcap y \hat{o} **in** order-trans)
apply (rule inf-le2)

```

apply simp
apply (rule gt-one-comp)
apply (subst dual-le, simp add: A)
apply (rule-tac y = y  $\hat{\ } o$  in order-trans)
apply (rule-tac y = y *  $\top \sqcap y \hat{\ } o$  in order-trans)
apply (rule inf-le2)
apply simp
apply (rule gt-one-comp)
by (subst dual-le, simp add: A)
also have ... = (( $x * \top$ )  $\sqcap$  ( $x \hat{\ } o$ )) * (( $y * \top$ )  $\sqcap$  ( $y \hat{\ } o$ ))
apply (cut-tac x = x in dual-assertion-conjunctive)
apply (cut-tac A, cut-tac B, simp add: assertion-def)
by (simp add: inf-comp conjunctiveD)
also have ... =  $x * y$ 
by (cut-tac B, cut-tac D, simp)
finally show  $x \sqcap y \leq x * y$  .
qed

```

lemma [*simp*]: $x \in \text{assertion} \implies x * y \leq y$
by (*unfold assertion-def, cut-tac x = x and y = 1 and z = y in le-comp-right, simp-all*)

lemma [*simp*]: $x \in \text{assertion} \implies y \in \text{assertion} \implies x * y \leq x$
apply (*subgoal-tac x * y \leq ($x * \top$) \sqcap ($x \hat{\ } o$)*)
apply (*simp add: assertion-def*)
apply (*simp, safe*)
apply (*rule le-comp, simp*)
apply (*rule-tac y = 1 in order-trans*)
apply (*rule-tac y = y in order-trans*)
apply *simp*
apply (*simp add: assertion-def*)
by (*subst dual-le, simp add: assertion-def*)

lemma *assertion-inf-comp-eq*: $x \in \text{assertion} \implies y \in \text{assertion} \implies x \sqcap y = x * y$
by (*rule order.antisym, simp-all*)

lemma *one-right-assertion* [*simp*]: $x \in \text{assertion} \implies x * 1 = x$
apply (*drule assertion-prop*)
proof –
assume $A: x * \top \sqcap 1 = x$
have $x * 1 = (x * \top \sqcap 1) * 1$ **by** (*simp add: A*)
also have ... = $x * \top \sqcap 1$ **by** (*simp add: inf-comp*)
also have ... = x **by** (*simp add: A*)
finally show *?thesis* .
qed

lemma [*simp*]: $x \in \text{assertion} \implies x \sqcup 1 = 1$
by (*rule order.antisym, simp-all add: assertion-def*)

lemma [simp]: $x \in \text{assertion} \implies 1 \sqcup x = 1$
 by (rule order.antisym, simp-all add: assertion-def)

lemma [simp]: $x \in \text{assertion} \implies x \sqcap 1 = x$
 by (rule order.antisym, simp-all add: assertion-def)

lemma [simp]: $x \in \text{assertion} \implies 1 \sqcap x = x$
 by (rule order.antisym, simp-all add: assertion-def)

lemma [simp]: $x \in \text{assertion} \implies x \leq x * \top$
 by (cut-tac $x = 1$ and $y = \top$ and $z = x$ in le-comp, simp-all)

lemma [simp]: $x \in \text{assertion} \implies x \leq 1$
 by (simp add: assertion-def)

definition

$\text{neg-assert } (x::'a) = (x \hat{ } o * \perp) \sqcap 1$

lemma sup-uminus[simp]: $x \in \text{assertion} \implies x \sqcup \text{neg-assert } x = 1$
 apply (simp add: neg-assert-def)
 apply (simp add: sup-inf-distrib)
 apply (rule order.antisym, simp-all)
 apply (unfold assertion-def)
 apply safe
 apply (subst dual-le)
 apply (simp add: dual-sup dual-comp)
 apply (subst inf-commute)
 by simp

lemma inf-uminus[simp]: $x \in \text{assertion} \implies x \sqcap \text{neg-assert } x = \perp$
 apply (simp add: neg-assert-def)
 apply (rule order.antisym, simp-all)
 apply (rule-tac $y = x \sqcap (x \hat{ } o * \perp)$ in order-trans)
 apply simp
 apply (rule-tac $y = x \hat{ } o * \perp \sqcap 1$ in order-trans)
 apply (rule inf-le2)
 apply simp
 apply (rule-tac $y = (x * \top) \sqcap (x \hat{ } o * \perp)$ in order-trans)
 apply simp
 apply (rule-tac $y = x$ in order-trans)
 apply simp-all
 by (simp add: dual-neg)

lemma uminus-assertion[simp]: $x \in \text{assertion} \implies \text{neg-assert } x \in \text{assertion}$
 apply (subst assertion-def)
 apply (simp add: neg-assert-def)

apply (*simp add: inf-comp dual-inf dual-comp inf-sup-distrib*)
apply (*subst inf-commute*)
by (*simp add: dual-neg*)

lemma *uminus-uminus* [*simp*]: $x \in \text{assertion} \implies \text{neg-assert } (\text{neg-assert } x) = x$
apply (*simp add: neg-assert-def*)
by (*simp add: dual-inf dual-comp sup-comp assertion-prop*)

lemma *dual-comp-neg* [*simp*]: $x \hat{o} * y \sqcup (\text{neg-assert } x) * \top = x \hat{o} * y$
apply (*simp add: neg-assert-def inf-comp*)
apply (*rule order.antisym, simp-all*)
by (*rule le-comp, simp*)

lemma [*simp*]: $(\text{neg-assert } x) \hat{o} * y \sqcup x * \top = (\text{neg-assert } x) \hat{o} * y$
apply (*simp add: neg-assert-def inf-comp dual-inf dual-comp sup-comp*)
by (*rule order.antisym, simp-all*)

lemma [*simp*]: $x * \top \sqcup (\text{neg-assert } x) \hat{o} * y = (\text{neg-assert } x) \hat{o} * y$
by (*simp add: neg-assert-def inf-comp dual-inf dual-comp sup-comp*)

lemma *inf-assertion* [*simp*]: $x \in \text{assertion} \implies y \in \text{assertion} \implies x \sqcap y \in \text{assertion}$
apply (*subst assertion-def*)
apply *safe*
apply (*rule-tac y = x in order-trans*)
apply *simp-all*
apply (*simp add: assertion-inf-comp-eq*)
proof –
assume *A*: $x \in \text{assertion}$
assume *B*: $y \in \text{assertion}$
have *C*: $(x * \top) \sqcap (x \hat{o}) = x$
by (*cut-tac A, unfold assertion-def, simp*)
have *D*: $(y * \top) \sqcap (y \hat{o}) = y$
by (*cut-tac B, unfold assertion-def, simp*)
have $x * y = ((x * \top) \sqcap (x \hat{o})) * ((y * \top) \sqcap (y \hat{o}))$ **by** (*simp add: C D*)
also have $\dots = x * \top \sqcap ((x \hat{o}) * ((y * \top) \sqcap (y \hat{o})))$ **by** (*simp add: inf-comp*)
also have $\dots = x * \top \sqcap ((x \hat{o}) * (y * \top)) \sqcap ((x \hat{o}) * (y \hat{o}))$
by (*cut-tac A, cut-tac x = x in dual-assertion-conjunctive, simp-all add: conjunctiveD inf-assoc*)
also have $\dots = (((x * \top) \sqcap (x \hat{o})) * (y * \top)) \sqcap ((x \hat{o}) * (y \hat{o}))$
by (*simp add: inf-comp*)
also have $\dots = (x * y * \top) \sqcap ((x * y) \hat{o})$ **by** (*simp add: C mult.assoc dual-comp*)
finally show $(x * y * \top) \sqcap ((x * y) \hat{o}) = x * y$ **by** *simp*
qed

lemma *comp-assertion* [*simp*]: $x \in \text{assertion} \implies y \in \text{assertion} \implies x * y \in \text{assertion}$
by (*subst assertion-inf-comp-eq [THEN sym], simp-all*)

lemma *sup-assertion* [*simp*]: $x \in \text{assertion} \implies y \in \text{assertion} \implies x \sqcup y \in \text{assertion}$
apply (*subst assertion-def*)
apply *safe*
apply (*unfold assertion-def*)
apply *simp*
apply *safe*
proof –
assume [*simp*]: $x \leq 1$
assume [*simp*]: $y \leq 1$
assume A : $x * \top \sqcap x \hat{\ } o = x$
assume B : $y * \top \sqcap y \hat{\ } o = y$
have $(y * \top) \sqcap (x \hat{\ } o) \sqcap (y \hat{\ } o) = (x \hat{\ } o) \sqcap (y * \top) \sqcap (y \hat{\ } o)$ **by** (*simp add: inf-commute*)
also have $\dots = (x \hat{\ } o) \sqcap ((y * \top) \sqcap (y \hat{\ } o))$ **by** (*simp add: inf-assoc*)
also have $\dots = (x \hat{\ } o) \sqcap y$ **by** (*simp add: B*)
also have $\dots = y$
apply (*rule order.antisym, simp-all*)
apply (*rule-tac y = 1 in order-trans*)
apply *simp*
by (*subst dual-le, simp*)
finally have [*simp*]: $(y * \top) \sqcap (x \hat{\ } o) \sqcap (y \hat{\ } o) = y$.
have $x * \top \sqcap (x \hat{\ } o) \sqcap (y \hat{\ } o) = x \sqcap (y \hat{\ } o)$ **by** (*simp add: A*)
also have $\dots = x$
apply (*rule order.antisym, simp-all*)
apply (*rule-tac y = 1 in order-trans*)
apply *simp*
by (*subst dual-le, simp*)
finally have [*simp*]: $x * \top \sqcap (x \hat{\ } o) \sqcap (y \hat{\ } o) = x$.
have $(x \sqcup y) * \top \sqcap (x \sqcup y) \hat{\ } o = (x * \top \sqcup y * \top) \sqcap ((x \hat{\ } o) \sqcap (y \hat{\ } o))$ **by**
(*simp add: sup-comp dual-sup*)
also have $\dots = x \sqcup y$ **by** (*simp add: inf-sup-distrib inf-assoc [THEN sym]*)
finally show $(x \sqcup y) * \top \sqcap (x \sqcup y) \hat{\ } o = x \sqcup y$.
qed

lemma [*simp*]: $x \in \text{assertion} \implies x * x = x$
by (*simp add: assertion-inf-comp-eq [THEN sym]*)

lemma [*simp*]: $x \in \text{assertion} \implies (x \hat{\ } o) * (x \hat{\ } o) = x \hat{\ } o$
apply (*rule dual-eq*)
by (*simp add: dual-comp assertion-inf-comp-eq [THEN sym]*)

lemma [*simp*]: $x \in \text{assertion} \implies x * (x \hat{\ } o) = x$
proof –
assume A : $x \in \text{assertion}$
have B : $x * \top \sqcap (x \hat{\ } o) = x$ **by** (*cut-tac A, unfold assertion-def, simp*)
have $x * x \hat{\ } o = (x * \top \sqcap (x \hat{\ } o)) * x \hat{\ } o$ **by** (*simp add: B*)
also have $\dots = x * \top \sqcap (x \hat{\ } o)$ **by** (*cut-tac A, simp add: inf-comp*)

also have $\dots = x$ **by** (*simp add: B*)
finally show *?thesis* .
qed

lemma [*simp*]: $x \in \text{assertion} \implies (x \hat{\ } o) * x = x \hat{\ } o$
apply (*rule dual-eq*)
by (*simp add: dual-comp*)

lemma [*simp*]: $\perp \in \text{assertion}$
by (*unfold assertion-def, simp*)

lemma [*simp*]: $1 \in \text{assertion}$
by (*unfold assertion-def, simp*)

3.2 Weakest precondition of true

definition

$wpt\ x = (x * \top) \sqcap 1$

lemma *wpt-is-assertion* [*simp*]: $wpt\ x \in \text{assertion}$
apply (*unfold wpt-def assertion-def, safe*)
apply *simp*
apply (*simp add: inf-comp dual-inf dual-comp inf-sup-distrib*)
apply (*rule order.antisym*)
by (*simp-all add: dual-neg*)

lemma *wpt-comp*: $(wpt\ x) * x = x$
apply (*simp add: wpt-def inf-comp*)
apply (*rule order.antisym, simp-all*)
by (*cut-tac x = 1 and y = \top and z = x in le-comp, simp-all*)

lemma *wpt-comp-2*: $wpt\ (x * y) = wpt\ (x * (wpt\ y))$
by (*simp add: wpt-def inf-comp mult.assoc*)

lemma *wpt-assertion* [*simp*]: $x \in \text{assertion} \implies wpt\ x = x$
by (*simp add: wpt-def assertion-prop*)

lemma *wpt-le-assertion*: $x \in \text{assertion} \implies x * y = y \implies wpt\ y \leq x$

apply (*simp add: wpt-def*)

proof –

assume *A*: $x \in \text{assertion}$

assume *B*: $x * y = y$

have $y * \top \sqcap 1 = x * (y * \top) \sqcap 1$ **by** (*simp add: B mult.assoc [THEN sym]*)

also have $\dots \leq x * \top \sqcap 1$

apply *simp*

apply (*rule-tac y = x * (y * \top) in order-trans*)

apply *simp-all*

by (*rule le-comp, simp*)

also have $\dots = x$ **by** (*cut-tac A, simp add: assertion-prop*)
finally show $y * \top \sqcap 1 \leq x$.
qed

lemma *wpt-choice*: $wpt (x \sqcap y) = wpt x \sqcap wpt y$
apply (*simp add: wpt-def inf-comp*)
proof –
have $x * \top \sqcap 1 \sqcap (y * \top \sqcap 1) = x * \top \sqcap ((y * \top \sqcap 1) \sqcap 1)$ **apply** (*subst inf-assoc*) **by** (*simp add: inf-commute*)
also have $\dots = x * \top \sqcap (y * \top \sqcap 1)$ **by** (*subst inf-assoc, simp*)
also have $\dots = (x * \top) \sqcap (y * \top) \sqcap 1$ **by** (*subst inf-assoc, simp*)
finally show $x * \top \sqcap (y * \top) \sqcap 1 = x * \top \sqcap 1 \sqcap (y * \top \sqcap 1)$ **by** *simp*
qed
end

context *lattice* **begin**
lemma [*simp*]: $x \leq y \implies x \sqcap y = x$
by (*simp add: inf-absorb1*)
end

context *mbt-algebra* **begin**

lemma *wpt-dual-assertion-comp*: $x \in \text{assertion} \implies y \in \text{assertion} \implies wpt ((x \hat{\ } o) * y) = (neg\text{-assert } x) \sqcup y$
apply (*simp add: wpt-def neg-assert-def*)
proof –
assume *A*: $x \in \text{assertion}$
assume *B*: $y \in \text{assertion}$
have *C*: $((x \hat{\ } o) * \perp) \sqcup 1 = x \hat{\ } o$
by (*rule dual-assertion-prop, rule A*)
have $x \hat{\ } o * y * \top \sqcap 1 = (((x \hat{\ } o) * \perp) \sqcup 1) * y * \top \sqcap 1$ **by** (*simp add: C*)
also have $\dots = ((x \hat{\ } o) * \perp \sqcup (y * \top)) \sqcap 1$ **by** (*simp add: sup-comp*)
also have $\dots = (((x \hat{\ } o) * \perp) \sqcap 1) \sqcup ((y * \top) \sqcap 1)$ **by** (*simp add: inf-sup-distrib2*)
also have $\dots = (((x \hat{\ } o) * \perp) \sqcap 1) \sqcup y$ **by** (*cut-tac B, drule assertion-prop, simp*)
finally show $x \hat{\ } o * y * \top \sqcap 1 = (((x \hat{\ } o) * \perp) \sqcap 1) \sqcup y$.
qed

lemma *le-comp-left-right*: $x \leq y \implies u \leq v \implies x * u \leq y * v$
apply (*rule-tac y = x * v in order-trans*)
apply (*rule le-comp, simp*)
by (*rule le-comp-right, simp*)

lemma *wpt-dual-assertion*: $x \in \text{assertion} \implies wpt (x \hat{\ } o) = 1$
apply (*simp add: wpt-def*)
apply (*rule order.antisym*)
apply *simp-all*

apply (*cut-tac* $x = 1$ **and** $y = x \hat{\ } o$ **and** $u = 1$ **and** $v = \top$ **in** *le-comp-left-right*)
apply *simp-all*
apply (*subst dual-le*)
by *simp*

lemma *assertion-commute*: $x \in \text{assertion} \implies y \in \text{conjunctive} \implies y * x = \text{wpt}(y * x) * y$
apply (*simp add: wpt-def*)
apply (*simp add: inf-comp*)
apply (*drule-tac* $x = y$ **and** $y = x * \top$ **and** $z = 1$ **in** *conjunctiveD*)
by (*simp add: mult.assoc [THEN sym] assertion-prop*)

lemma *wpt-mono*: $x \leq y \implies \text{wpt } x \leq \text{wpt } y$
apply (*simp add: wpt-def*)
apply (*rule-tac* $y = x * \top$ **in** *order-trans, simp-all*)
by (*rule le-comp-right, simp*)

lemma $a \in \text{conjunctive} \implies x * a \leq a * y \implies (x \hat{\ } \omega) * a \leq a * (y \hat{\ } \omega)$
apply (*rule omega-least*)
apply (*simp add: mult.assoc [THEN sym]*)
apply (*rule-tac* $y = a * y * y \hat{\ } \omega \sqcap a$ **in** *order-trans*)
apply (*simp*)
apply (*rule-tac* $y = x * a * y \hat{\ } \omega$ **in** *order-trans, simp-all*)
apply (*rule le-comp-right, simp*)
apply (*simp add: mult.assoc*)
apply (*subst (2) omega-fix*)
by (*simp add: conjunctiveD*)

lemma [*simp*]: $x \leq 1 \implies y * x \leq y$
by (*cut-tac* $x = x$ **and** $y = 1$ **and** $z = y$ **in** *le-comp, simp-all*)

lemma [*simp*]: $x \leq x * \top$
by (*cut-tac* $x = 1$ **and** $y = \top$ **and** $z = x$ **in** *le-comp, simp-all*)

lemma [*simp*]: $x * \perp \leq x$
by (*cut-tac* $x = \perp$ **and** $y = 1$ **and** $z = x$ **in** *le-comp, simp-all*)

end

3.3 Monotonic Boolean transformers algebra with post condition statement

definition

post-fun ($p::'a::\text{order}$) $q = (\text{if } p \leq q \text{ then } (\top::'b::\{\text{order-bot}, \text{order-top}\}) \text{ else } \perp)$

lemma *mono-post-fun* [*simp*]: $\text{mono } (\text{post-fun } (p::-\::\{\text{order-bot}, \text{order-top}\}))$
apply (*simp add: post-fun-def mono-def, safe*)
apply (*subgoal-tac* $p \leq y$, *simp*)

```

apply (rule-tac  $y = x$  in order-trans)
apply simp-all
done

lemma post-top [simp]: post-fun  $p$   $p = \top$ 
by (simp add: post-fun-def)

lemma post-refin [simp]: mono  $S \implies ((S\ p)::'a::bounded-lattice) \sqcap (post-fun\ p)\ x \leq S\ x$ 
apply (simp add: le-fun-def assert-fun-def post-fun-def, safe)
by (rule-tac  $f = S$  in monoD, simp-all)

class post-mbt-algebra = mbt-algebra +
  fixes post :: 'a  $\Rightarrow$  'a
  assumes post-1: (post  $x$ ) *  $x * \top = \top$ 
  and post-2:  $y * x * \top \sqcap (post\ x) \leq y$ 

instantiation MonoTran :: (complete-boolean-algebra) post-mbt-algebra
begin

lift-definition post-MonoTran :: 'a::complete-boolean-algebra MonoTran  $\Rightarrow$  'a::complete-boolean-algebra MonoTran
is  $\lambda x.$  post-fun ( $x\ \top$ )
by (rule mono-post-fun)

instance proof
fix  $x :: 'a$  MonoTran show post  $x * x * \top = \top$ 
  apply transfer
  apply (simp add: fun-eq-iff)
  done
fix  $x\ y :: 'a$  MonoTran show  $y * x * \top \sqcap post\ x \leq y$ 
  apply transfer
  apply (simp add: le-fun-def)
  done
qed

end

```

3.4 Complete monotonic Boolean transformers algebra

```

class complete-mbt-algebra = post-mbt-algebra + complete-distrib-lattice +
  assumes Inf-comp: (Inf  $X$ ) *  $z = (INF\ x \in X. (x * z))$ 

instance MonoTran :: (complete-boolean-algebra) complete-mbt-algebra
apply intro-classes
apply transfer
apply (simp add: Inf-comp-fun)
done

```

context *complete-mbt-algebra* **begin**

lemma *dual-Inf*: $(\text{Inf } X) \hat{o} = (\text{SUP } x \in X . x \hat{o})$

apply (*rule order.antisym*)
apply (*subst dual-le, simp*)
apply (*rule Inf-greatest*)
apply (*subst dual-le, simp*)
apply (*rule SUP-upper, simp*)
apply (*rule SUP-least*)
apply (*subst dual-le, simp*)
by (*rule Inf-lower, simp*)

lemma *dual-Sup*: $(\text{Sup } X) \hat{o} = (\text{INF } x \in X . x \hat{o})$

apply (*rule order.antisym*)
apply (*rule INF-greatest*)
apply (*subst dual-le, simp*)
apply (*rule Sup-upper, simp*)
apply (*subst dual-le, simp*)
apply (*rule Sup-least*)
apply (*subst dual-le, simp*)
by (*rule INF-lower, simp*)

lemma *INF-comp*: $(\prod (f \text{ ' } A)) * z = (\text{INF } a \in A . (f a) * z)$

unfolding *Inf-comp*
apply (*subgoal-tac* $((\lambda x::'a. x * z) \text{ ' } f \text{ ' } A) = ((\lambda a::'b. f a * z) \text{ ' } A)$)
by *auto*

lemma *dual-INF*: $(\prod (f \text{ ' } A)) \hat{o} = (\text{SUP } a \in A . (f a) \hat{o})$

unfolding *Inf-comp dual-Inf*
apply (*subgoal-tac* $(\text{dual ' } f \text{ ' } A) = ((\lambda a::'b. f a \hat{o}) \text{ ' } A)$)
by *auto*

lemma *dual-SUP*: $(\bigsqcup (f \text{ ' } A)) \hat{o} = (\text{INF } a \in A . (f a) \hat{o})$

unfolding *dual-Sup*
apply (*subgoal-tac* $(\text{dual ' } f \text{ ' } A) = ((\lambda a::'b. f a \hat{o}) \text{ ' } A)$)
by *auto*

lemma *Sup-comp*: $(\text{Sup } X) * z = (\text{SUP } x \in X . (x * z))$

apply (*rule dual-eq*)
by (*simp add: dual-comp dual-Sup dual-SUP INF-comp image-comp*)

lemma *SUP-comp*: $(\bigsqcup (f \text{ ' } A)) * z = (\text{SUP } a \in A . (f a) * z)$

unfolding *Sup-comp*
apply (*subgoal-tac* $((\lambda x::'a. x * z) \text{ ' } f \text{ ' } A) = ((\lambda a::'b. f a * z) \text{ ' } A)$)
by *auto*

lemma *Sup-assertion [simp]*: $X \subseteq \text{assertion} \implies \text{Sup } X \in \text{assertion}$

apply (*unfold assertion-def*)
apply *safe*

```

apply (rule Sup-least)
apply blast
apply (simp add: Sup-comp dual-Sup Sup-inf)
apply (subgoal-tac (( $\lambda y . y \sqcap \sqcap (dual \text{ ' } X)$ ) ' ( $\lambda x . x * \top$ ) '  $X$ ) =  $X$ )
apply simp
proof -
  assume  $A: X \subseteq \{x. x \leq 1 \wedge x * \top \sqcap x \hat{\ } o = x\}$ 
  have  $B$  [simp]: !!  $x . x \in X \implies x * \top \sqcap (\sqcap (dual \text{ ' } X)) = x$ 
  proof -
    fix  $x$ 
    assume  $C: x \in X$ 
    have  $x * \top \sqcap \sqcap (dual \text{ ' } X) = x * \top \sqcap (x \hat{\ } o \sqcap \sqcap (dual \text{ ' } X))$ 
      apply (subgoal-tac  $\sqcap (dual \text{ ' } X) = (x \hat{\ } o \sqcap \sqcap (dual \text{ ' } X))$ , simp)
      apply (rule order.antisym, simp-all)
      apply (rule Inf-lower, cut-tac C, simp)
    done
    also have  $\dots = x \sqcap \sqcap (dual \text{ ' } X)$  by (unfold inf-assoc [THEN sym], cut-tac
A, cut-tac C, auto)
    also have  $\dots = x$ 
      apply (rule order.antisym, simp-all)
      apply (rule INF-greatest)
      apply (cut-tac A C)
      apply (rule-tac  $y = 1$  in order-trans)
      apply auto[1]
      apply (subst dual-le, auto)
    done
    finally show  $x * \top \sqcap \sqcap (dual \text{ ' } X) = x .$ 
  qed
  show ( $\lambda y . y \sqcap \sqcap (dual \text{ ' } X)$ ) ' ( $\lambda x . x * \top$ ) '  $X = X$ 
    by (simp add: image-comp)
qed

```

```

lemma Sup-range-assertion [simp]: (!! $w . p w \in assertion$ )  $\implies Sup (range p) \in$ 
assertion
  by (rule Sup-assertion, auto)

```

```

lemma Sup-less-assertion [simp]: (!! $w . p w \in assertion$ )  $\implies Sup-less p w \in as-$ 
sertion
  by (unfold Sup-less-def, rule Sup-assertion, auto)

```

```

theorem omega-lfp:
   $x \hat{\ } \omega * y = lfp (\lambda z . (x * z) \sqcap y)$ 
  apply (rule order.antisym)
  apply (rule lfp-greatest)
  apply (drule omega-least, simp)
  apply (rule lfp-lowerbound)
  apply (subst (2) omega-fix)
  by (simp add: inf-comp mult.assoc)
end

```

```

lemma [simp]: mono ( $\lambda (t::'a::\text{mbt-algebra}) . x * t \sqcap y$ )
  apply (simp add: mono-def, safe)
  apply (rule-tac y = x * xa in order-trans, simp)
  by (rule le-comp, simp)

```

```

class mbt-algebra-fusion = mbt-algebra +
  assumes fusion: ( $\forall t . x * t \sqcap y \sqcap z \leq u * (t \sqcap z) \sqcap v$ )
     $\implies (x \hat{\omega}) * y \sqcap z \leq (u \hat{\omega}) * v$ 

```

```

lemma
  class.mbt-algebra-fusion (1::'a::complete-mbt-algebra) ((*) (sqcap) (le) (<) (sqcup) dual
dual-star omega star bot top
  apply unfold-locales
  apply (cut-tac h = lambda t . t sqcap z and f = lambda t . x * t sqcap y and g = lambda t . u * t sqcap
v in weak-fusion)
  apply (rule inf-Disj)
  apply simp-all
  apply (simp add: le-fun-def)
  by (simp add: omega-lfp)

```

```

context mbt-algebra-fusion
begin

```

```

lemma omega-star:  $x \in \text{conjunctive} \implies x \hat{\omega} = \text{wpt } (x \hat{\omega}) * (x \hat{*})$ 
  apply (simp add: wpt-def inf-comp)
  apply (rule order.antisym)
  apply (cut-tac x = x and y = 1 and z = x hat{omega} * top sqcap x hat{*} in omega-least)
  apply (simp-all add: conjunctiveD,safe)
  apply (subst (2) omega-fix)
  apply (simp add: inf-comp inf-assoc mult.assoc)
  apply (metis inf.commute inf-assoc inf-le1 star-fix)
  apply (cut-tac x = x and y = top and z = x hat{*} and u = x and v = 1 in
fusion)
  apply (simp add: conjunctiveD)
  apply (metis inf-commute inf-le1 le-infE star-fix)
  by (metis mult.right-neutral)

```

```

lemma omega-pres-conj:  $x \in \text{conjunctive} \implies x \hat{\omega} \in \text{conjunctive}$ 
  apply (subst omega-star, simp)
  apply (rule comp-pres-conj)
  apply (rule assertion-conjunctive, simp)
  by (rule start-pres-conj, simp)
end

```

```

end

```


4 Boolean Algebra of Assertions

```

theory Assertion-Algebra
imports Mono-Bool-Tran-Algebra
begin

```

This section introduces the boolean algebra of assertions. The type `Assertion` and the boolean operation are introduced based on the set `assertion` and the operations on the monotonic boolean transformers algebra. The type `Assertion` over a complete monotonic boolean transformers algebra is a complete boolean algebra.

```

typedef (overloaded) ('a::mbt-algebra) Assertion = assertion::'a set
apply (rule-tac  $x = \perp$  in exI)
by (unfold assertion-def, simp)

```

definition

```

assert :: 'a::mbt-algebra Assertion  $\Rightarrow$  'a ( $\{\cdot -\}$  [0] 1000) where
 $\{\cdot p\} = \text{Rep-Assertion } p$ 

```

definition

```

abs-wpt  $x = \text{Abs-Assertion } (\text{wpt } x)$ 

```

lemma [*simp*]: $\{\cdot p\} \in \text{assertion}$

```

by (unfold assert-def, cut-tac  $x = p$  in Rep-Assertion, simp)

```

lemma [*simp*]: $\text{abs-wpt } (\{\cdot p\}) = p$

```

apply (simp add: abs-wpt-def)
by (simp add: assert-def Rep-Assertion-inverse)

```

lemma [*simp*]: $x \in \text{assertion} \Longrightarrow \{\cdot \text{Abs-Assertion } x\} = x$

```

apply (simp add: assert-def)
by (rule Abs-Assertion-inverse, simp)

```

lemma [*simp*]: $x \in \text{assertion} \Longrightarrow \{\cdot \text{abs-wpt } x\} = x$

```

apply (simp add: abs-wpt-def assert-def)
by (rule Abs-Assertion-inverse, simp)

```

lemma *assert-injective*: $\{\cdot p\} = \{\cdot q\} \Longrightarrow p = q$

proof –

```

assume  $A: \{\cdot p\} = \{\cdot q\}$ 
have  $p = \text{abs-wpt } (\{\cdot p\})$  by simp
also have  $\dots = q$  by (subst  $A$ , simp)
finally show ?thesis .

```

qed

instantiation *Assertion* :: (*mbt-algebra*) *boolean-algebra*

begin

definition

uminus-Assertion-def: $\neg p = \text{abs-wpt}(\text{neg-assert } \{ \cdot p \})$

definition

bot-Assertion-def: $\perp = \text{abs-wpt } \perp$

definition

top-Assertion-def: $\top = \text{abs-wpt } 1$

definition

inf-Assertion-def: $p \sqcap q = \text{abs-wpt} (\{ \cdot p \} \sqcap \{ \cdot q \})$

definition

sup-Assertion-def: $p \sqcup q = \text{abs-wpt} (\{ \cdot p \} \sqcup \{ \cdot q \})$

definition

less-eq-Assertion-def: $(p \leq q) = (\{ \cdot p \} \leq \{ \cdot q \})$

definition

less-Assertion-def: $(p < q) = (\{ \cdot p \} < \{ \cdot q \})$

definition

minus-Assertion-def: $(p :: 'a \text{ Assertion}) - q = p \sqcap \neg q$

instance**proof**

fix $x y :: 'a \text{ Assertion}$ **show** $(x < y) = (x \leq y \wedge \neg y \leq x)$

by (*simp add: less-Assertion-def less-eq-Assertion-def less-le-not-le*)

next

fix $x :: 'a \text{ Assertion}$ **show** $x \leq x$ **by** (*simp add: less-eq-Assertion-def*)

next

fix $x y z :: 'a \text{ Assertion}$ **assume** $A: x \leq y$ **assume** $B: y \leq z$ **from** A **and** B

show $x \leq z$

by (*simp add: less-eq-Assertion-def*)

next

fix $x y :: 'a \text{ Assertion}$ **assume** $A: x \leq y$ **assume** $B: y \leq x$ **from** A **and** B

show $x = y$

apply (*cut-tac p = x and q = y in assert-injective*)

by (*rule antisym, simp-all add: less-eq-Assertion-def*)

next

fix $x y :: 'a \text{ Assertion}$ **show** $x \sqcap y \leq x$

by (*simp add: less-eq-Assertion-def inf-Assertion-def*)

fix $x y :: 'a \text{ Assertion}$ **show** $x \sqcap y \leq y$

by (*simp add: less-eq-Assertion-def inf-Assertion-def*)

next

fix $x y z :: 'a \text{ Assertion}$ **assume** $A: x \leq y$ **assume** $B: x \leq z$ **from** A **and** B

show $x \leq y \sqcap z$

by (*simp add: less-eq-Assertion-def inf-Assertion-def*)

next

```

    fix x y :: 'a Assertion show  $x \leq x \sqcup y$ 
      by (simp add: less-eq-Assertion-def sup-Assertion-def)
    fix x y :: 'a Assertion show  $y \leq x \sqcup y$ 
      by (simp add: less-eq-Assertion-def sup-Assertion-def)
  next
    fix x y z :: 'a Assertion assume A:  $y \leq x$  assume B:  $z \leq x$  from A and B
  show  $y \sqcup z \leq x$ 
    by (simp add: less-eq-Assertion-def sup-Assertion-def)
  next
    fix x :: 'a Assertion show  $\perp \leq x$ 
      by (simp add: less-eq-Assertion-def bot-Assertion-def)
  next
    fix x :: 'a Assertion show  $x \leq \top$ 
      by (simp add: less-eq-Assertion-def top-Assertion-def)
  next
    fix x y z :: 'a Assertion show  $x \sqcup y \sqcap z = (x \sqcup y) \sqcap (x \sqcup z)$ 
      by (simp add: less-eq-Assertion-def sup-Assertion-def inf-Assertion-def sup-inf-distrib)
  next
    fix x :: 'a Assertion show  $x \sqcap - x = \perp$ 
      by (simp add: inf-Assertion-def uminus-Assertion-def bot-Assertion-def)
  next
    fix x :: 'a Assertion show  $x \sqcup - x = \top$ 
      by (simp add: sup-Assertion-def uminus-Assertion-def top-Assertion-def)
  next
    fix x y :: 'a Assertion show  $x - y = x \sqcap - y$ 
      by (simp add: minus-Assertion-def)
  qed
end

```

lemma *assert-image [simp]: assert 'A* \subseteq *assertion*
 by *auto*

instantiation *Assertion* :: (*complete-mbt-algebra*) *complete-lattice*
 begin

definition

Sup-Assertion-def: Sup A = *abs-wpt (Sup (assert 'A))*

definition

Inf-Assertion-def: Inf (A::('a Assertion) set) = $-$ (*Sup (uminus 'A)*)

lemma *Sup1: (x::'a Assertion) \in A* $\implies x \leq$ *Sup A*

apply (*simp add: Sup-Assertion-def less-eq-Assertion-def Abs-Assertion-inverse*)
 by (*rule Sup-upper, simp*)

lemma *Sup2: ($\bigwedge x::'a Assertion . x \in A \implies x \leq z$)* \implies *Sup A* $\leq z$

apply (*simp add: Sup-Assertion-def less-eq-Assertion-def Abs-Assertion-inverse*)
apply (*rule Sup-least*)

```

    by blast

instance
proof
  fix x :: 'a Assertion fix A assume A: x ∈ A from A show Inf A ≤ x
    apply (simp add: Inf-Assertion-def)
    apply (subst compl-le-compl-iff [THEN sym], simp)
    by (rule Sup1, simp)
next
  fix z :: 'a Assertion fix A assume A:  $\bigwedge x . x \in A \implies z \leq x$  from A show z ≤
  Inf A
    apply (simp add: Inf-Assertion-def)
    apply (subst compl-le-compl-iff [THEN sym], simp)
    apply (rule Sup2)
    apply safe
    by simp
next
  fix x :: 'a Assertion fix A assume A: x ∈ A from A show x ≤ Sup A
    by (rule Sup1)
next
  fix z :: 'a Assertion fix A assume A:  $\bigwedge x . x \in A \implies x \leq z$  from A show Sup
  A ≤ z
    by (rule Sup2)
next
  show Inf {} = ( $\top$ ::'a Assertion)
    by (auto simp: Inf-Assertion-def Sup-Assertion-def compl-bot-eq [symmetric]
  bot-Assertion-def)
next
  show Sup {} = ( $\perp$ ::'a Assertion)
    by (auto simp: Sup-Assertion-def bot-Assertion-def)
qed
end

lemma assert-top [simp]: { $\cdot$ . $\top$ } = 1
  by (simp add: top-Assertion-def)

lemma assert-Sup: { $\cdot$ .Sup A} = Sup (assert ' A)
  by (simp add: Sup-Assertion-def Abs-Assertion-inverse)

lemma assert-Inf: { $\cdot$ .Inf A} = (Inf (assert ' A))  $\sqcap$  1
proof (cases A = {})
  case True then show ?thesis by simp
next
  note image-cong-simp [cong del]
  case False then show ?thesis
    apply (simp add: Inf-Assertion-def uminus-Assertion-def)
    apply (simp add: neg-assert-def assert-Sup dual-Sup Inf-comp inf-commute inf-Inf
  comp-def)

```

```

apply (rule-tac f = Inf in arg-cong)
apply safe
apply simp
apply (subst inf-commute)
apply (simp add: image-def uminus-Assertion-def)
apply (simp add: neg-assert-def dual-comp dual-inf sup-comp assertion-prop)
apply auto [1]
apply (simp)
apply (subst image-def, simp)
apply (simp add: uminus-Assertion-def)
apply (subst inf-commute)
apply (simp add: neg-assert-def dual-comp dual-inf sup-comp assertion-prop)
apply auto
done
qed

```

```

lemma assert-Inf-ne:  $A \neq \{\}$   $\implies \{\cdot \text{Inf } A\} = \text{Inf } (\text{assert } 'A)$ 
apply (unfold assert-Inf)
apply (rule antisym)
apply simp-all
apply safe
apply (erule notE)
apply (rule-tac y =  $\{x\}$  in order-trans)
by (simp-all add: INF-lower)

```

```

lemma assert-Sup-range:  $\{\cdot \text{Sup } (\text{range } p)\} = \text{Sup } (\text{range } (\text{assert } o p))$ 
apply (subst assert-Sup)
by (rule-tac f = Sup in arg-cong, auto)

```

```

lemma assert-Sup-less:  $\{\cdot \text{Sup-less } p w\} = \text{Sup-less } (\text{assert } o p) w$ 
apply (simp add: Sup-less-def)
apply (subst assert-Sup)
by (rule-tac f = Sup in arg-cong, auto)

```

end

5 Program statements, Hoare and refinement rules

```

theory Statements
imports Assertion-Algebra
begin

```

In this section we introduce assume, if, and while program statements as well as Hoare triples, and data refinement. We prove Hoare correctness rules for the program statements and we prove some theorems linking Hoare correctness statement to (data) refinement. Most of the theorems assume a monotonic boolean transformers algebra. The theorem stating the equiva-

lence between a Hoare correctness triple and a refinement statement holds under the assumption that we have a monotonic boolean transformers algebra with post condition statement.

definition

assume :: 'a::mbt-algebra Assertion \Rightarrow 'a ([· -] [0] 1000) **where**
 $[\cdot p] = \{\cdot p\} \hat{\ } o$

lemma [simp]: $\{\cdot p\} * \top \sqcap [\cdot p] = \{\cdot p\}$
apply (subgoal-tac $\{\cdot p\} \in \text{assertion}$)
apply (subst (asm) assertion-def, simp add: assume-def)
by simp

lemma [simp]: $[\cdot p] * x \sqcup \{\cdot \neg p\} * \top = [\cdot p] * x$
by (simp add: assume-def uminus-Assertion-def)

lemma [simp]: $\{\cdot p\} * \top \sqcup [\cdot \neg p] * x = [\cdot \neg p] * x$
by (simp add: assume-def uminus-Assertion-def)

lemma assert-sup: $\{\cdot p \sqcup q\} = \{\cdot p\} \sqcup \{\cdot q\}$
by (simp add: sup-Assertion-def)

lemma assert-inf: $\{\cdot p \sqcap q\} = \{\cdot p\} \sqcap \{\cdot q\}$
by (simp add: inf-Assertion-def)

lemma assert-neg: $\{\cdot \neg p\} = \text{neg-assert } \{\cdot p\}$
by (simp add: uminus-Assertion-def)

lemma assert-false [simp]: $\{\cdot \perp\} = \perp$
by (simp add: bot-Assertion-def)

lemma if-Assertion-assumption: $(\{\cdot p\} * x) \sqcup (\{\cdot \neg p\} * y) = ([\cdot p] * x) \sqcap ([\cdot \neg p] * y)$

proof –

have $(\{\cdot p\} * x) \sqcup \{\cdot \neg p\} * y = (\{\cdot p\} * \top \sqcap [\cdot p]) * x \sqcup (\{\cdot \neg p\} * \top \sqcap [\cdot \neg p]) * y$ **by** simp

also have $\dots = (\{\cdot p\} * \top \sqcap ([\cdot p] * x)) \sqcup (\{\cdot \neg p\} * \top \sqcap ([\cdot \neg p] * y))$ **by** (unfold inf-comp, simp)

also have $\dots = ((\{\cdot p\} * \top \sqcap ([\cdot p] * x)) \sqcup (\{\cdot \neg p\} * \top)) \sqcap ((\{\cdot p\} * \top \sqcap ([\cdot p] * x)) \sqcup ([\cdot \neg p] * y))$ **by** (simp add: sup-inf-distrib)

also have $\dots = ((\{\cdot p\} * \top \sqcup (\{\cdot \neg p\} * \top)) \sqcap (([\cdot p] * x))) \sqcap (([\cdot \neg p] * y) \sqcap (([\cdot p] * x) \sqcup ([\cdot \neg p] * y)))$

by (simp add: sup-inf-distrib2)

also have $\dots = ([\cdot p] * x) \sqcap ([\cdot \neg p] * y) \sqcap (([\cdot p] * x) \sqcup ([\cdot \neg p] * y))$

apply (simp add: sup-comp [THEN sym])

by (simp add: assert-sup [THEN sym] inf-assoc)

also have $\dots = ([\cdot p] * x) \sqcap ([\cdot \neg p] * y)$

by (rule antisym, simp-all add: inf-assoc)

finally show ?thesis .

qed

definition

$$wp\ x\ p = abs-wpt\ (x * \{\cdot p\})$$
lemma *wp-assume*: $wp\ [\cdot p]\ q = \neg p \sqcup q$
apply (*simp add: wp-def abs-wpt-def*)

apply (*rule assert-injective*)

apply *simp*
by (*simp add: assert-sup assert-neg assume-def wpt-dual-assertion-comp*)

lemma *assert-commute*: $y \in conjunctive \implies y * \{\cdot p\} = \{\cdot wp\ y\ p\} * y$
apply (*simp add: wp-def abs-wpt-def*)

by (*rule assertion-commute, simp-all*)

lemma *wp-assert*: $wp\ \{\cdot p\}\ q = p \sqcap q$
by (*simp add: wp-def assertion-inf-comp-eq [THEN sym] assert-inf [THEN sym]*)

lemma *wp-mono [simp]*: $mono\ (wp\ x)$
apply (*simp add: le-fun-def wp-def abs-wpt-def less-eq-Assertion-def mono-def*)

apply (*simp add: wpt-def, safe*)

apply (*rule-tac y = x * \{\cdot xa\} * \top* **in** *order-trans, simp-all*)

apply (*rule le-comp-right*)

by (*rule le-comp, simp*)

lemma *wp-mono2*: $p \leq q \implies wp\ x\ p \leq wp\ x\ q$
apply (*cut-tac x = x* **in** *wp-mono*)

apply (*unfold mono-def*)

by *blast*
lemma *wp-fun-mono [simp]*: $mono\ wp$
apply (*simp add: le-fun-def wp-def abs-wpt-def less-eq-Assertion-def mono-def*)

apply (*simp add: wpt-def, safe*)

apply (*rule-tac y = x * \{\cdot xa\} * \top* **in** *order-trans, simp-all*)

apply (*rule le-comp-right*)

by (*rule le-comp-right, simp*)

lemma *wp-fun-mono2*: $x \leq y \implies wp\ x\ p \leq wp\ y\ p$
apply (*cut-tac wp-fun-mono*)

apply (*unfold mono-def*)

apply (*simp add: le-fun-def*)

by *blast*
lemma *wp-comp*: $wp\ (x * y)\ p = wp\ x\ (wp\ y\ p)$
apply (*simp add: wp-def abs-wpt-def*)

by (*unfold wp-comp-2 [THEN sym] mult.assoc, simp*)

lemma *wp-choice*: $wp\ (x \sqcap y) = wp\ x \sqcap wp\ y$

apply (*simp add: fun-eq-iff wp-def inf-fun-def inf-comp inf-Assertion-def abs-wpt-def*)
by (*simp add: wpt-choice*)

lemma [*simp*]: $wp\ 1 = id$
apply (*unfold fun-eq-iff, safe*)
apply (*rule assert-injective*)
by (*simp add: wp-def abs-wpt-def*)

lemma *wp-omega-fix*: $wp\ (x \hat{\omega})\ p = wp\ x\ (wp\ (x \hat{\omega})\ p) \sqcap p$
apply (*subst omega-fix*)
by (*simp add: wp-choice wp-comp*)

lemma *wp-omega-least*: $(wp\ x\ r) \sqcap p \leq r \implies wp\ (x \hat{\omega})\ p \leq r$
apply (*simp add: wp-def abs-wpt-def inf-Assertion-def less-eq-Assertion-def*)
apply (*simp add: wpt-def*)
apply (*rule-tac y = {r} * \top \sqcap 1 in order-trans*)
apply *simp*
apply (*rule-tac y = x $\hat{\omega}$ * {p} * \top in order-trans, simp*)
apply (*simp add: mult.assoc*)
apply (*rule omega-least*)
apply (*drule-tac z = \top in le-comp-right*)
apply (*simp add: inf-comp mult.assoc [THEN sym]*)
by (*simp add: assertion-prop*)

lemma *Assertion-wp*: $\{ \cdot wp\ x\ p \} = (x * \{ \cdot p \} * \top) \sqcap 1$
apply (*simp add: wp-def abs-wpt-def*)
by (*simp add: wpt-def*)

definition
 $hoare\ p\ S\ q = (p \leq wp\ S\ q)$

definition
 $grd\ x = - (wp\ x\ \perp)$

lemma *grd-comp*: $[\cdot\ grd\ x] * x = x$
apply (*simp add: grd-def wp-def uminus-Assertion-def assume-def neg-assert-def*
abs-wpt-def dual-sup sup-comp)
apply (*simp add: wpt-def dual-inf sup-comp dual-comp bot-Assertion-def*)
by (*rule antisym, simp-all*)

lemma *assert-assume*: $\{ \cdot p \} * [\cdot p] = \{ \cdot p \}$
by (*simp add: assume-def*)

lemma *dual-assume*: $[\cdot p] \hat{o} = \{ \cdot p \}$
by (*simp add: assume-def*)

lemma *assume-prop*: $([\cdot p] * \perp) \sqcup 1 = [\cdot p]$
by (*simp add: assume-def dual-assertion-prop*)

An alternative definition of a Hoare triple

definition $hoare1\ p\ S\ q = ([\cdot\ p] * S * [\cdot\ -q] = \top)$

lemma $hoare1\ p\ S\ q = hoare\ p\ S\ q$

apply (*simp add: hoare1-def dual-inf dual-comp*)

apply (*simp add: hoare-def wp-def less-eq-Assertion-def abs-wpt-def*)

apply (*simp add: wpt-def*)

apply *safe*

proof –

assume $A: [\cdot\ p] * S * [\cdot\ -q] = \top$

have $\{\cdot\ p\} \leq \{\cdot\ p\} * \top$ **by** *simp*

also have $\dots \leq \{\cdot\ p\} * \top * \perp$ **by** (*unfold mult.assoc, simp*)

also have $\dots = \{\cdot\ p\} * [\cdot\ p] * S * [\cdot\ -q] * \perp$ **by** (*subst A [THEN sym], simp add: mult.assoc*)

also have $\dots = \{\cdot\ p\} * S * [\cdot\ -q] * \perp$ **by** (*simp add: assert-assume*)

also have $\dots \leq \{\cdot\ p\} * S * \{\cdot\ q\} * \top$

apply (*simp add: mult.assoc*)

apply (*rule le-comp, rule le-comp*)

apply (*simp add: assume-def uminus-Assertion-def*)

by (*simp add: neg-assert-def dual-inf dual-comp sup-comp*)

also have $\dots \leq S * \{\cdot\ q\} * \top$ **by** (*simp add: mult.assoc*)

finally show $\{\cdot\ p\} \leq S * \{\cdot\ q\} * \top$.

next

assume $A: \{\cdot\ p\} \leq S * \{\cdot\ q\} * \top$

have $\top = ((S * \{\cdot\ q\}) \wedge o) * \perp \sqcup S * \{\cdot\ q\} * \top$ **by** *simp*

also have $\dots \leq [\cdot\ p] * \perp \sqcup S * \{\cdot\ q\} * \top$

apply (*simp del: dual-neg-top*)

apply (*rule-tac y = [\cdot\ p] * \perp in order-trans, simp-all*)

apply (*subst dual-le*)

apply (*simp add: dual-comp dual-assume*)

apply (*cut-tac x = \{\cdot\ p\} and y = S * \{\cdot\ q\} * \top and z = \top in le-comp-right*)

apply (*rule A*)

by (*simp add: mult.assoc*)

also have $\dots = [\cdot\ p] * S * (\{\cdot\ q\} * \top)$

apply (*subst (2) assume-prop [THEN sym]*)

by (*simp-all add: sup-comp mult.assoc*)

also have $\dots \leq [\cdot\ p] * S * (\{\cdot\ q\} * \top \sqcup 1)$

by (*rule le-comp, simp*)

also have $\dots = [\cdot\ p] * S * [\cdot\ -q]$

apply (*simp add: assume-def uminus-Assertion-def*)

by (*simp add: neg-assert-def dual-inf dual-comp*)

finally show $[\cdot\ p] * S * [\cdot\ -q] = \top$

by (*rule-tac antisym, simp-all*)

qed

lemma *hoare-choice*: $hoare\ p\ (x \sqcap y)\ q = ((hoare\ p)\ x\ q \ \&\ (hoare\ p)\ y\ q)$

apply (*unfold hoare-def wp-choice inf-fun-def*)

by *auto*

definition

if-stm:: 'a::mbt-algebra Assertion \Rightarrow 'a \Rightarrow 'a \Rightarrow 'a ((If (-)/ then (-)/ else (-)) [0, 0, 10] 10) **where**

if-stm b x y = (([· b] * x) \sqcap ([· -b] * y))

lemma *if-assertion*: (If p then x else y) = {·p} * x \sqcup {·-p} * y
by (*simp add: if-stm-def if-Assertion-assumption*)

lemma *hoare-if*: hoare p (If b then x else y) q = (hoare (p \sqcap b) x q \wedge hoare (p \sqcap -b) y q)

by (*simp add: hoare-def if-stm-def wp-choice inf-fun-def wp-comp wp-assume sup-neg-inf*)

lemma *hoare-comp*: hoare p (x * y) q = (\exists r . (hoare p x r) \wedge (hoare r y q))

apply (*simp add: hoare-def wp-comp*)

apply *safe*

apply (*rule-tac x = wp y q in exI, simp*)

apply (*rule-tac y = wp x r in order-trans, simp*)

apply (*rule-tac f = wp x in monoD*)

by *simp-all*

lemma *hoare-refinement*: hoare p S q = ({·p} * (post {·q}) \leq S)

apply (*simp add: hoare-def less-eq-Assertion-def Assertion-wp*)

proof

assume A: {·p} \leq S * {·q} * \top

have {·p} * post {·q} = ({·p} * \top \sqcap 1) * post {·q} **by** (*simp add: assertion-prop*)

also have ... = {·p} * \top \sqcap post {·q} **by** (*simp add: inf-comp*)

also have ... \leq S * {·q} * \top \sqcap post {·q} **apply** *simp*

apply (*rule-tac y = {·p} * \top in order-trans, simp-all*)

apply (*cut-tac x = {·p} and y = S * {·q} * \top and z = \top in le-comp-right*)

by (*rule A, simp*)

also have ... \leq S **by** (*simp add: post-2*)

finally show {·p} * post {·q} \leq S.

next

assume A: {·p} * post {·q} \leq S

have {·p} = {·p} * \top \sqcap 1 **by** (*simp add: assertion-prop*)

also have ... = {·p} * ((post {·q}) * {·q} * \top) \sqcap 1 **by** (*simp add: post-1*)

also have ... \leq {·p} * ((post {·q}) * {·q} * \top) **by** *simp*

also have ... \leq S * {·q} * \top

apply (*cut-tac x = {·p} * post {·q} and y = S and z = {·q} * \top in le-comp-right*)

apply (*simp add: A*)

by (*simp add: mult.assoc*)

finally show {·p} \leq S * {·q} * \top .

qed

theorem *hoare-fixpoint-mbt*:

$F x = x$

\implies (! (w::'a::well-founded) f . ($\bigwedge v. v < w \implies$ hoare (p v) f q) \implies hoare (p w) (F f) q)

$\implies \text{hoare } (p \ u) \ x \ q$
apply (*rule less-induct1*)
proof –
fix xa
assume $A: \bigwedge w \ f. (\bigwedge v . v < w \implies \text{hoare } (p \ v) \ f \ q) \implies \text{hoare } (p \ w) \ (F \ f) \ q$
assume $B: F \ x = x$
assume $C: \bigwedge y . y < xa \implies \text{hoare } (p \ y) \ x \ q$
have $D: \text{hoare } (p \ xa) \ (F \ x) \ q$
apply (*rule A*)
by (*rule C, simp*)
show $\text{hoare } (p \ xa) \ x \ q$
by (*cut-tac D, simp add: B*)
qed

lemma *hoare-Sup*: $\text{hoare } (\text{Sup } P) \ x \ q = (\forall p \in P . \text{hoare } p \ x \ q)$
apply (*simp add: hoare-def*)
apply *auto*
apply (*rule-tac y = Sup P in order-trans, simp-all add: Sup-upper*)
apply (*rule Sup-least*)
by *simp*

theorem *hoare-fixpoint-complete-mbt*:
 $F \ x = x$
 $\implies (!! \ w \ f . \text{hoare } (\text{Sup-less } p \ w) \ f \ q \implies \text{hoare } (p \ w) \ (F \ f) \ q)$
 $\implies \text{hoare } (\text{Sup } (\text{range } p)) \ x \ q$
apply (*simp add: hoare-Sup Sup-less-def, safe*)
apply (*rule-tac F = F in hoare-fixpoint-mbt*)
by *auto*

definition

$\text{while} :: 'a :: \text{mbt-algebra} \ \text{Assertion} \Rightarrow 'a \Rightarrow 'a \ ((\text{While } (-) / \text{do } (-)) \ [0, 10] \ 10)$
where
 $\text{while } p \ x = ([\cdot \ p] * x) \wedge^\omega * [\cdot \ -p]$

lemma *while-false*: $(\text{While } \perp \ \text{do } x) = 1$
apply (*unfold while-def*)
apply (*subst omega-fix*)
by (*simp-all add: assume-def*)

lemma *while-true*: $(\text{While } \top \ \text{do } 1) = \perp$
apply (*unfold while-def*)
by (*rule antisym, simp-all add: assume-def*)

lemma *hoare-wp* [*simp*]: $\text{hoare } (wp \ x \ q) \ x \ q$
by (*simp add: hoare-def*)

lemma *hoare-comp-wp*: $\text{hoare } p \ (x * y) \ q = \text{hoare } p \ x \ (wp \ y \ q)$
apply (*unfold hoare-comp, safe*)
apply (*simp add: hoare-def*)

apply (*rule-tac* $y = wp\ x\ r$ **in** *order-trans*, *simp*)
apply (*rule* *wp-mono2*, *simp*)
by (*rule-tac* $x = wp\ y\ q$ **in** *exI*, *simp*)

lemma (**in** *mbt-algebra*) *hoare-assume*: $hoare\ p\ [\cdot b]\ q = (p \sqcap b \leq q)$
by (*simp* *add*: *hoare-def* *wp-assume* *sup-neg-inf*)

lemma (**in** *mbt-algebra*) *hoare-assume-comp*: $hoare\ p\ ([\cdot b] * x)\ q = hoare\ (p \sqcap b)\ x\ q$
apply (*simp* *add*: *hoare-comp-wp* *hoare-assume*)
by (*simp* *add*: *hoare-def*)

lemma *hoare-while-mbt*:
 $(\forall\ w::'b::well-founded)\ r . (\forall\ v . v < w \longrightarrow p\ v \leq r) \longrightarrow hoare\ ((p\ w) \sqcap b)\ x\ r) \Longrightarrow$
 $(\forall\ u . p\ u \leq q) \Longrightarrow hoare\ (p\ w)\ (While\ b\ do\ x)\ (q \sqcap -b)$
apply (*unfold* *while-def*)
apply (*rule-tac* $F = \lambda z. [\cdot b] * x * z \sqcap [\cdot - b]$ **in** *hoare-fixpoint-mbt*)
apply (*simp* *add*: *mult.assoc* [*THEN* *sym*])
apply (*simp* *add*: *omega-comp-fix*)
apply (*unfold* *hoare-choice*)
apply *safe*
apply (*subst* *hoare-comp-wp*)
apply (*subst* *hoare-assume-comp*)
apply (*drule-tac* $x = w$ **in** *spec*)
apply (*drule-tac* $x = wp\ f\ (q \sqcap -b)$ **in** *spec*)
apply (*auto* *simp* *add*: *hoare-def*) [1]
apply (*auto* *simp* *add*: *hoare-assume*)
apply (*rule-tac* $y = p\ w$ **in** *order-trans*)
by *simp-all*

lemma *hoare-while-complete-mbt*:
 $(\forall\ w::'b::well-founded . hoare\ ((p\ w) \sqcap b)\ x\ (Sup-less\ p\ w)) \Longrightarrow$
 $hoare\ (Sup\ (range\ p))\ (While\ b\ do\ x)\ ((Sup\ (range\ p)) \sqcap -b)$
apply (*simp* *add*: *hoare-Sup*, *safe*)
apply (*rule* *hoare-while-mbt*)
apply *safe*
apply (*drule-tac* $x = w$ **in** *spec*)
apply (*simp* *add*: *hoare-def*)
apply (*rule-tac* $y = wp\ x\ (Sup-less\ p\ w)$ **in** *order-trans*, *simp-all*)
apply (*rule* *wp-mono2*)
apply (*simp* *add*: *Sup-less-def*)
apply (*rule* *Sup-least*, *auto*)
by (*rule* *SUP-upper*, *simp*)

definition

$datarefin\ S\ S1\ D\ D1 = (D * S \leq S1 * D1)$

lemma $hoare\ p\ S\ q \Longrightarrow datarefin\ S\ S1\ D\ D1 \Longrightarrow hoare\ (wp\ D\ p)\ S1\ (wp\ D1\ q)$

apply (*simp add: hoare-def datarefin-def*)
apply (*simp add: wp-comp [THEN sym] mult.assoc [THEN sym]*)
apply (*rule-tac y = wp (D * S) q in order-trans*)
apply (*subst wp-comp*)
apply (*rule monoD, simp-all*)
by (*rule wp-fun-mono2, simp-all*)

lemma *hoare p S q \implies datarefin ($\{\cdot p\} * S$) S1 D D1 \implies hoare (wp D p) S1 (wp D1 q)*

apply (*simp add: hoare-def datarefin-def*)
apply (*rule-tac y = wp (D * $\{\cdot p\} * S$) q in order-trans*)
apply (*simp add: mult.assoc*)
apply (*subst wp-comp*)
apply (*rule monoD, simp-all*)
apply (*subst wp-comp*)
apply (*unfold wp-assert, simp*)
apply (*unfold wp-comp [THEN sym]*)
apply (*rule wp-fun-mono2*)
by (*simp add: mult.assoc*)

lemma *inf-pres-conj: x \in conjunctive \implies y \in conjunctive \implies x \sqcap y \in conjunctive*

apply (*subst conjunctive-def, safe*)
apply (*simp add: inf-comp conjunctiveD*)
by (*metis (opaque-lifting, no-types) inf-assoc inf-left-commute*)

lemma *sup-pres-disj: x \in disjunctive \implies y \in disjunctive \implies x \sqcup y \in disjunctive*

apply (*subst disjunctive-def, safe*)
apply (*simp add: sup-comp disjunctiveD*)
by (*metis (opaque-lifting, no-types) sup-assoc sup-left-commute*)

lemma *assumption-conjunctive [simp]: $[\cdot p] \in$ conjunctive*

by (*simp add: assume-def dual-disjunctive assertion-disjunctive*)

lemma *assumption-disjunctive [simp]: $[\cdot p] \in$ disjunctive*

by (*simp add: assume-def dual-conjunctive assertion-conjunctive*)

lemma *if-pres-conj: x \in conjunctive \implies y \in conjunctive \implies (If p then x else y) \in conjunctive*

apply (*unfold if-stm-def*)
by (*simp add: inf-pres-conj comp-pres-conj*)

lemma *if-pres-disj: x \in disjunctive \implies y \in disjunctive \implies (If p then x else y) \in disjunctive*

apply (*unfold if-assertion*)
by (*simp add: sup-pres-disj comp-pres-disj assertion-disjunctive*)

lemma *while-dual-star: (While p do (x::'a::mbt-algebra)) = (($\{\cdot p\} * x$) $\widehat{\otimes}$ * $\{\cdot -p\}$)*

apply (*simp add: while-def*)

```

apply (rule antisym)
apply (rule omega-least)
proof -
  have  $([\cdot p] * x * ((\{\cdot p\} * x) \hat{\otimes} * \{\cdot - p\}) \sqcap [\cdot - p]) = (\{\cdot p\} * x * ((\{\cdot p\} * x) \hat{\otimes} * \{\cdot - p\})) \sqcup \{\cdot - p\}$ 
    apply (unfold mult.assoc)
    by (cut-tac p = p and x = (x * ((\{\cdot p\} * x) \hat{\otimes} * \{\cdot - p\})) and y = 1 in if-Assertion-assumption, simp)
    also have  $\dots = (\{\cdot p\} * x) \hat{\otimes} * \{\cdot - p\}$ 
      by (simp add: mult.assoc [THEN sym], simp add: dual-star-comp-fix [THEN sym])
    finally show  $[\cdot p] * x * ((\{\cdot p\} * x) \hat{\otimes} * \{\cdot - p\}) \sqcap [\cdot - p] \leq (\{\cdot p\} * x) \hat{\otimes} * \{\cdot - p\}$  by simp
  next
    show  $(\{\cdot p\} * x) \hat{\otimes} * \{\cdot - p\} \leq ([\cdot p] * x) \hat{\omega} * [\cdot - p]$ 
      apply (rule dual-star-least)
      proof -
        have  $\{\cdot p\} * x * (([\cdot p] * x) \hat{\omega} * [\cdot - p]) \sqcup \{\cdot - p\} = [\cdot p] * x * (([\cdot p] * x) \hat{\omega} * [\cdot - p]) \sqcap [\cdot - p]$ 
          apply (unfold mult.assoc)
          by (cut-tac p = p and x = (x * (([\cdot p] * x) \hat{\omega} * [\cdot - p])) and y = 1 in if-Assertion-assumption, simp)
          also have  $\dots = ([\cdot p] * x) \hat{\omega} * [\cdot - p]$ 
            apply (simp add: mult.assoc [THEN sym])
            by (metis omega-comp-fix)
          finally show  $\{\cdot p\} * x * (([\cdot p] * x) \hat{\omega} * [\cdot - p]) \sqcup \{\cdot - p\} \leq ([\cdot p] * x) \hat{\omega} * [\cdot - p]$  by simp
        qed
      qed

```

lemma *while-pres-disj*: $(x::'a::\text{mbt-algebra}) \in \text{disjunctive} \implies (\text{While } p \text{ do } x) \in \text{disjunctive}$

```

apply (unfold while-dual-star)
apply (rule comp-pres-disj)
apply (rule dual-star-pres-disj)
by (rule comp-pres-disj, simp-all add: assertion-disjunctive)

```

lemma *while-pres-conj*: $(x::'a::\text{mbt-algebra-fusion}) \in \text{conjunctive} \implies (\text{While } p \text{ do } x) \in \text{conjunctive}$

```

apply (unfold while-def)
by (simp add: comp-pres-conj omega-pres-conj)

```

unbundle *no-lattice-syntax*

end

References

- [1] R.-J. Back. *On the correctness of refinement in program development*. PhD thesis, Department of Computer Science, University of Helsinki, 1978.
- [2] R.-J. Back. *Correctness preserving program refinements: proof theory and applications*, volume 131 of *Mathematical Centre Tracts*. Mathematisch Centrum, Amsterdam, 1980.
- [3] R.-J. Back and J. von Wright. A lattice-theoretical basis for a specification language. In *Proceedings of the International Conference on Mathematics of Program Construction, 375th Anniversary of the Groningen University*, pages 139–156, London, UK, 1989. Springer-Verlag.
- [4] R.-J. Back and J. von Wright. Duality in specification languages: a lattice-theoretical approach. *Acta Inf.*, 27:583–625, July 1990.
- [5] R.-J. Back and J. von Wright. *Refinement Calculus. A systematic Introduction*. Springer, 1998.
- [6] H.-H. Dang, P. Höfner, and B. Möller. Algebraic separation logic. *Journal of Logic and Algebraic Programming*, 80(6):221 – 247, 2011. Relations and Kleene Algebras in Computer Science.
- [7] J. Desharnais, B. Möller, and G. Struth. Kleene algebra with domain. *ACM Trans. Comput. Logic*, 7:798–833, October 2006.
- [8] P. Guerreiro. Another characterization of weakest preconditions. In M. Dezani-Ciancaglini and U. Montanari, editors, *International Symposium on Programming*, volume 137 of *Lecture Notes in Computer Science*, pages 164–177. Springer Berlin / Heidelberg, 1982. 10.1007/3-540-11494-7_12.
- [9] C. A. Hoare, B. Möller, G. Struth, and I. Wehrman. Concurrent kleene algebra. In *Proceedings of the 20th International Conference on Concurrency Theory, CONCUR 2009*, pages 399–414, Berlin, Heidelberg, 2009. Springer-Verlag.
- [10] D. Kozen. Kleene algebra with tests. *ACM Trans. Program. Lang. Syst.*, 19:427–443, May 1997.
- [11] L. Meinicke and K. Solin. Refinement algebra for probabilistic programs. *Formal Aspects of Computing*, 22:3–31, 2010. 10.1007/s00165-009-0111-1.
- [12] C. Morgan. *Programming from specifications*. Prentice-Hall, Inc., 1990.

- [13] V. Preoteasa. Algebra of monotonic boolean transformers. In *Proceedings of SBMF 2011*, Lecture Notes in Computer Science, pages 140–155, Berlin Heidelberg, 2011. Springer-Verlag.
- [14] K. Solin and J. von Wright. Enabledness and termination in refinement algebra. *Sci. Comput. Program.*, 74:654–668, June 2009.
- [15] J. von Wright. From kleene algebra to refinement algebra. In *Proceedings of the 6th International Conference on Mathematics of Program Construction*, MPC '02, pages 233–262, London, UK, UK, 2002. Springer-Verlag.
- [16] J. von Wright. Towards a refinement algebra. *Sci. Comput. Program.*, 51:23–45, May 2004.