Algebra of Monotonic Boolean Transformers

Viorel Preoteasa

March 17, 2025

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

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 $((\mathsf{dual}\ f)p = -f(-p))$ 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*:
$\bigsqcap M \circ f = (\bigsqcap m{\in}M.\ m \circ f)$
by (*simp add*: *fun-eq-iff image-comp*)

lemma *INF-comp-fun*:
$(\bigsqcap a{\in}A.\ g\ a) \circ f = (\bigsqcap a{\in}A.\ g\ a \circ f)$
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 $(\lambda\text{-}.\ c)$
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 $f \Longrightarrow$ *mono* $g \Longrightarrow$ *mono* $(f \circ g)$
by (*auto intro*!: *monoI elim*!: *monoE order-class.monoE*)

lemma (**in** *bot*) *mono-bot* [*simp*]:
mono $\bot$
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 ‹*mono f*› ‹$a \leq b$› **have** $\ldots \leq f\ b$ **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 ‹*mono g*› ‹$a \leq b$› **have** $\ldots \leq g\ b$ **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 ‹*mono f*› ‹$a \leq b$› **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 ‹*mono g*› ‹$a \leq b$› **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 $A \subseteq \{f :: 'a \Rightarrow 'b :: \textit{complete-lattice}.\ \textit{mono}\ f\}$
shows *mono* $(\bigsqcap A)$
proof
fix *a b*
assume $a \leq b$
{ fix *f*
assume $f \in A$
with *assms* **have** *mono f* **by** *auto*
with ‹$a \leq b$› **have** $f\ a \leq f\ b$ **by** (*auto elim*: *monoE*)
}
then have $(\bigsqcap f \in A.\ f\ a) \leq (\bigsqcap f \in A.\ f\ b)$
by (*auto intro*: *complete-lattice-class.INF-greatest complete-lattice-class.INF-lower2*)
then show $(\bigsqcap A)\ a \leq (\bigsqcap A)\ b$ **by** *simp*
qed

lemma (**in** *complete-lattice*) *mono-Sup* [*simp*]:
  assumes $A \subseteq \{f :: 'a \Rightarrow 'b :: complete\text{-}lattice.\ mono\ f\}$
  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**) $'a\ MonoTran = \{f :: 'a :: order \Rightarrow 'a\ .\ mono\ f\}$
proof
  show $id \in ?MonoTran$ **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 $\bot$
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* ⇒ *'a MonoTran*
is *Inf*
by (*rule mono-Inf*) *auto*

lift-definition *Sup-MonoTran* :: *'a MonoTran set* ⇒ *'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* ===> (*A* ===> *pcr-MonoTran HOL.eq*) ===> *pcr-MonoTran HOL.eq*) (λ*A f*. ⨅(*f* ‘ *A*)) (λ*A f*. ⨅(*f* ‘ *A*))
by *transfer-prover*

lemma [*transfer-rule*]:
(*rel-set A* ===> (*A* ===> *pcr-MonoTran HOL.eq*) ===> *pcr-MonoTran HOL.eq*) (λ*A f*. ⨆(*f* ‘ *A*)) (λ*A f*. ⨆(*f* ‘ *A*))
by *transfer-prover*

end

instance *MonoTran* :: (*complete-distrib-lattice*) *complete-distrib-lattice*
proof (*intro-classes*, *transfer*)
fix *A* :: (*'a* ⇒ *'a*) *set set*
assume ∀*A*∈*A*. *Ball A mono*
from *this* **have** [*simp*]: {*f* ‘ *A* |*f*. ∀ *Y*∈*A*. *f Y* ∈ *Y*} = {*x*. (∃*f*. (∀ *x*. (∀ *x*∈*x*. *mono x*) ⟶ *mono* (*f x*)) ∧ *x* = *f* ‘ *A* ∧ (∀ *Y*∈*A*. *f Y* ∈ *Y*)) ∧ (∀ *x*∈*x*. *mono x*)}
apply *safe*
apply (*rule-tac x* = λ *x* . *if x* ∈ *A then f x else* ⊥ **in** *exI*)
apply (*simp add*: *if-split image-def*)
by *blast*+

show ⨅(*Sup* ‘ *A*) ≤ ⨆(*Inf* ‘ {*x*. (∃*f*∈*Collect* (*pred-fun* (λ*A*. *Ball A mono*) *mono*). *x* = *f* ‘ *A* ∧ (∀ *Y*∈*A*. *f Y* ∈ *Y*)) ∧ *Ball x mono*})
by (*simp add*: *Inf-Sup*)
qed

definition
dual-fun (*f*::*'a*::*boolean-algebra* ⇒ *'a*) = *uminus* ∘ *f* ∘ *uminus*

lemma *dual-fun-apply* [*simp*]:
dual-fun f p = − *f* (− *p*)
by (*simp add*: *dual-fun-def*)

lemma *mono-dual-fun* [*simp*]:
mono f ⟹ *mono* (*dual-fun f*)
apply (*rule monoI*)
apply (*erule monoE*)
apply *auto*
done

lemma (**in** *order*) *mono-inf-fun* [*simp*]:
fixes *x* :: *'b*::*semilattice-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-sup*
shows *mono* (*sup x*)
by (*auto intro*!: *order-class.monoI semilattice-sup-class.sup-mono*)

lemma *mono-comp-fun*:
fixes *f* :: *'a*::*order* ⇒ *'b*::*order*
shows *mono f* ⟹ *mono* ((∘) *f*)
by (*rule monoI*) (*auto simp add*: *le-fun-def elim*: *monoE*)

definition
Omega-fun f g = *inf g* ∘ *comp f*

lemma *Omega-fun-apply* [*simp*]:
Omega-fun f g h p = (*g p* ⊓ *f* (*h p*))
by (*simp add*: *Omega-fun-def*)

lemma *mono-Omega-fun* [*simp*]:
mono f ⟹ *mono* (*Omega-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* ⇒ *'a*::*semilattice-inf* **and** *g* :: *'c*::*semilattice-inf* ⇒ *'a*
shows *mono f* ⟹ *mono g* ⟹ *mono-mono* (*Omega-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 $\Longrightarrow$ $A \leq$ *Omega-fun f g A* $\Longrightarrow$ $A \leq$ *star-fun f* $\circ$ *g*
apply (*simp add*: *star-fun-def*)
apply (*rule-tac P* = λ *x* . $A \leq x \circ g$ **and** f = *Omega-fun f 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 $\bigwedge u$. *Omega-fun f g u* $\leq u \Longrightarrow A \leq u$
shows $A \leq$ *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 $\bigwedge u$. $u \leq$ *Omega-fun f g u* $\Longrightarrow u \leq A$
shows *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 $\Longrightarrow$ *omega-fun f* $\circ$ *g* = *lfp* (*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 = Omega-fun f g x* **in** *order-trans*)
apply *simp-all*
apply (*rule-tac f = 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*:
mono f $\Longrightarrow$ *star-fun f* $\circ$ *g* = *gfp* (*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 = Omega-fun f g x* **in** *order-trans*)
apply *simp-all*
apply (*rule-tac f = Omega-fun f g* **in** *monoD*)
apply *simp-all*
apply (*rule Sup-upper*)
by *simp*

definition
assert-fun p q = (*p* $\sqcap$ *q* :: *'a*::*semilattice-inf*)

lemma *mono-assert-fun* [*simp*]:
mono (*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*]: *assert-fun p* $\leq$ *id*
by (*simp add*: *assert-fun-def id-def le-fun-def*)

lemma *assert-fun-disjunctive* [*simp*]: *assert-fun* (*p*::*'a*::*distrib-lattice*) $\in$ *Apply.disjunctive*
by (*simp add*: *assert-fun-def Apply.disjunctive-def inf-sup-distrib*)

definition
assertion-fun = *range 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*)) = *assert-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* ⇒ *'a MonoTran*
is *dual-fun*
by (*fact mono-dual-fun*)

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

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

definition *dual-star-MonoTran* :: *'a MonoTran* ⇒ *'a MonoTran*
where
(*x*::(*'a MonoTran*)) ^⊗ = ((*x* ^ *o*) ^ ∗) ^ *o*

instance proof
fix *x y* :: *'a MonoTran* **show** (*x* ≤ *y*) = (*y* ^ *o* ≤ *x* ^ *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* ^ *o*) ^ *o* = *x*
apply *transfer*
apply (*simp add*: *fun-eq-iff*)
done
next
fix *x y* :: *'a MonoTran* **show** (*x* ∗ *y*) ^ *o* = *x* ^ *o* ∗ *y* ^ *o*
apply *transfer*
apply (*simp add*: *fun-eq-iff*)
done
next
show (*1*::*'a MonoTran*) ^ *o* = *1*
apply *transfer*
apply (*simp add*: *fun-eq-iff*)
done
next
fix *x* :: *'a MonoTran* **show** ⊤ ∗ *x* = ⊤
apply *transfer*
apply (*simp add*: *fun-eq-iff*)
done
next
fix *x y z* :: *'a MonoTran* **show** (*x* ⊓ *y*) ∗ *z* = (*x* ∗ *z*) ⊓ (*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*]: $\bot \frown o = \top$
apply (*rule order.antisym*, *simp-all*)
by (*subst dual-le*, *simp*)

lemma *dual-inf*: $(x \sqcap y) \frown o = (x \frown o) \sqcup (y \frown o)$
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*: $(x \sqcup y) \frown o = (x \frown o) \sqcap (y \frown o)$
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* $((x \frown o \sqcap y \frown o) * z \frown o) \frown o = ((x \frown o * z \frown o) \sqcap (y \frown o * z \frown o)) \frown o$)
apply (*simp add*: *dual-inf dual-comp*)
by (*simp add*: *inf-comp*)

lemma *dual-eq*: $x \frown o = y \frown o \Longrightarrow x = y$
apply (*subgoal-tac* $(x \frown o) \frown o = (y \frown o) \frown o$)
apply (*subst* (*asm*) *dual-dual*)
apply (*subst* (*asm*) *dual-dual*)
by *simp-all*

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

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

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

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

lemma *gt-one-comp*: $1 \leq x \Longrightarrow y \leq x * y$
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*: $(x * z) \sqcup y \leq z \Longrightarrow (x\hat{}\otimes) * y \leq z$
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*]: $1 \hat{} \omega = \bot$
apply (*rule order.antisym*, *simp-all*)
by (*cut-tac* $x = 1 :: 'a$ **and** $y = 1$ **and** $z = \bot$ **in** *omega-least*, *simp-all*)

lemma *omega-mono*: $x \leq y \Longrightarrow x \hat{} \omega \leq y \hat{} \omega$
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*: $x \in conjunctive \Longrightarrow x \hat{} o \in disjunctive$
apply (*simp add*: *conjunctive-def disjunctive-def*)
apply *safe*
apply (*rule dual-eq*)
by (*simp add*: *dual-comp dual-sup*)

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

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

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

lemma *start-pres-conj*: $x \in \mathit{conjunctive} \Longrightarrow (x \mathbin{\hat{}} *) \in \mathit{conjunctive}$
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*: $x \in \mathit{disjunctive} \Longrightarrow x\hat{}\otimes \in \mathit{disjunctive}$
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 progrm 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 $\bot$ and 1. In this formalization assertions can be defined much simpler using the dual operator.

definition
$\mathit{assertion} = \{x \,.\, x \leq 1 \wedge (x * \top) \sqcap (x \mathbin{\hat{}} o) = x\}$

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

```
    assume A: x * ⊤ ⊓ x ⌢ o = x
    have x * ⊤ ⊓ 1 ≤ x * ⊤ ⊓ x ⌢ o
      apply simp
      apply (rule-tac y = 1 in order-trans)
      apply simp
      apply (subst dual-le)
      by simp
    also have ... = x by (cut-tac A, simp)
    finally show x * ⊤ ⊓ 1 ≤ x .
  next
    assume A: x * ⊤ ⊓ x ⌢ o = x
    have x = x * ⊤ ⊓ x ⌢ o by (simp add: A)
    also have ... ≤ x * ⊤ by simp
    finally show x ≤ x * ⊤ .
  qed

lemma dual-assertion-prop: x ∈ assertion ⟹ ((x ⌢ o) * ⊥) ⊔ 1 = x ⌢ o
  apply (rule dual-eq)
  by (simp add: dual-sup dual-comp assertion-prop)

lemma assertion-disjunctive: x ∈ assertion ⟹ x ∈ disjunctive
  apply (simp add: disjunctive-def, safe)
  apply (drule assertion-prop)
  proof −
    assume A: x * ⊤ ⊓ 1 = x
    fix y z::'a
    have x * (y ⊔ z) = (x * ⊤ ⊓ 1) * (y ⊔ z) by (cut-tac  A, simp)
    also have ... = (x * ⊤) ⊓ (y ⊔ z) by (simp add: inf-comp)
    also have ... = ((x * ⊤) ⊓ y) ⊔ ((x * ⊤) ⊓ z) by (simp add: inf-sup-distrib)
     also have ... = (((x * ⊤) ⊓ 1) * y) ⊔ (((x * ⊤) ⊓ 1) * z) by (simp add:
inf-comp)
    also have ... = x * y ⊔ x * z by (cut-tac  A, simp)
    finally show x * (y ⊔ z) = x * y ⊔ x * z .
  qed

lemma Abs-MonoTran-injective: mono x ⟹ mono y ⟹ Abs-MonoTran x =
Abs-MonoTran y ⟹ 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: Rep-MonoTran ‘ disjunctive = Apply.disjunctive
  apply (simp add: disjunctive-def Apply.disjunctive-def)
  apply transfer
  apply auto
  proof −
    fix f :: 'a ⇒ 'a and a b
```

```
    assume prem: ∀ y. mono y ⟶ (∀ z. mono z ⟶ f ∘ y ⊔ z = (f ∘ y) ⊔ (f ∘ z))
    { fix g h :: 'b ⇒ 'a
      assume mono g and mono h
      then have f ∘ g ⊔ h = (f ∘ g) ⊔ (f ∘ h)
        using prem by blast
    } note ∗ = this
    assume mono f
    show f (a ⊔ b) = f a ⊔ f b (is ?P = ?Q)
    proof (rule order-antisym)
      show ?P ≤ ?Q
        using ∗ [of λ-. a λ-. b] by (simp add: comp-def fun-eq-iff)
    next
      from ‹mono f› show ?Q ≤ ?P
        using Fun.semilattice-sup-class.mono-sup by blast
    qed
  next
    fix f :: 'a ⇒ 'a
    assume ∀ y z. f (y ⊔ z) = f y ⊔ f z
    then have ∗: ⋀y z. f (y ⊔ z) = f y ⊔ f z by blast
    show mono f
    proof
      fix a b :: 'a
      assume a ≤ b
      then show f a ≤ 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 ∈ assertion ⟹ x ∈ conjunctive
  apply (simp add: conjunctive-def, safe)
  apply (drule assertion-prop)
  proof −
    assume A: x ∗ ⊤ ⊓ 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 $\ldots = (x * \top) \sqcap (y \sqcap z)$ **by** (*simp add*: *inf-comp*)
also have $\ldots = ((x * \top) \sqcap y) \sqcap ((x * \top) \sqcap z)$
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 $\ldots = (((x * \top) \sqcap 1) * y) \sqcap (((x * \top) \sqcap 1) * z)$ **by** (*simp add*: *inf-comp*)
also have $\ldots = (x * y) \sqcap (x * z)$ **by** (*cut-tac A*, *simp*)
finally show $x * (y \sqcap z) = (x * y) \sqcap (x * z)$ **.**
qed

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

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

lemma [*simp*]: $x \in assertion \Longrightarrow y \in assertion \Longrightarrow x \sqcap y \leq x * y$
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 $\ldots \leq (x * \top) \sqcap (((x\hat{}o) * (y * \top)) \sqcap ((x\hat{}o) * (y\hat{}o)))$
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 ⌢ o in order-trans)
    apply (rule-tac y = y * ⊤ ⊓ y ⌢ 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 * ⊤) ⊓ (x ⌢ o)) * ((y * ⊤) ⊓ (y ⌢ 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 ⊓ y ≤ x * y .
 qed

lemma [simp]: x ∈ assertion ⟹ x * y ≤ y
 by (unfold assertion-def, cut-tac x = x and y = 1 and z = y in le-comp-right,
simp-all)


lemma [simp]: x ∈ assertion ⟹ y ∈ assertion ⟹ x * y ≤ x
 apply (subgoal-tac x * y ≤ (x * ⊤) ⊓ (x ⌢ 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 ∈ assertion ⟹ y ∈ assertion ⟹ x ⊓ y = x * y
 by (rule order.antisym, simp-all)

lemma one-right-assertion [simp]: x ∈ assertion ⟹ x * 1 = x
 apply (drule assertion-prop)
 proof −
   assume A: x * ⊤ ⊓ 1 = x
   have x * 1 = (x * ⊤ ⊓ 1) * 1 by (simp add: A)
   also have ... = x * ⊤ ⊓ 1 by (simp add: inf-comp)
   also have ... = x by (simp add: A)
   finally show ?thesis .
 qed

lemma [simp]: x ∈ assertion ⟹ x ⊔ 1 = 1
 by (rule order.antisym, simp-all add: assertion-def)
```

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

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

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

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

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

definition
$\mathit{neg\text{-}assert}\ (x::'a) = (x\ \widehat{}\ o * \bot) \sqcap 1$

lemma *sup-uminus*[*simp*]: $x \in \mathit{assertion} \Longrightarrow x \sqcup \mathit{neg\text{-}assert}\ x = 1$
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*]: $x \in \mathit{assertion} \Longrightarrow x \sqcap \mathit{neg\text{-}assert}\ x = \bot$
apply (*simp add*: *neg-assert-def*)
apply (*rule order.antisym*, *simp-all*)
apply (*rule-tac* $y = x \sqcap (x\ \widehat{}\ o * \bot)$ **in** *order-trans*)
apply *simp*
apply (*rule-tac* $y = x\ \widehat{}\ o * \bot \sqcap 1$ **in** *order-trans*)
apply (*rule inf-le2*)
apply *simp*
apply (*rule-tac* $y = (x * \top)\ \sqcap (x\ \widehat{}\ o * \bot)$ **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*]: $x \in \mathit{assertion} \Longrightarrow \mathit{neg\text{-}assert}\ x \in \mathit{assertion}$
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*]: $x \in assertion \Longrightarrow neg\text{-}assert\ (neg\text{-}assert\ x) = x$
apply (*simp add*: *neg-assert-def*)
by (*simp add*: *dual-inf dual-comp sup-comp assertion-prop*)

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

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

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

lemma *inf-assertion* [*simp*]: $x \in assertion \Longrightarrow y \in assertion \Longrightarrow x \sqcap y \in assertion$
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*: $x \in assertion$
assume *B*: $y \in assertion$
have *C*: $(x * \top) \sqcap (x \frown o) = x$
by (*cut-tac A*, *unfold assertion-def*, *simp*)
have *D*: $(y * \top) \sqcap (y \frown o) = y$
by (*cut-tac B*, *unfold assertion-def*, *simp*)
have $x * y = ((x * \top) \sqcap (x \frown o)) * ((y * \top) \sqcap (y \frown o))$ **by** (*simp add*: *C D*)
also have $\ldots = x * \top \sqcap ((x \frown o) * ((y * \top) \sqcap (y \frown o)))$ **by** (*simp add*: *inf-comp*)
also have $\ldots = x * \top \sqcap ((x \frown o) * (y * \top)) \sqcap ((x \frown o) * (y \frown o))$
by (*cut-tac A*, *cut-tac* $x = x$ **in** *dual-assertion-conjunctive*, *simp-all add*: *conjunctiveD inf-assoc*)
also have $\ldots = (((x * \top) \sqcap (x \frown o)) * (y * \top)) \sqcap ((x \frown o) * (y \frown o))$
by (*simp add*: *inf-comp*)
also have $\ldots = (x * y * \top) \sqcap ((x * y) \frown o)$ **by** (*simp add*: *C mult.assoc dual-comp*)
finally show $(x * y * \top) \sqcap ((x * y) \frown o) = x * y$ **by** *simp*
qed

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

lemma *sup-assertion* [*simp*]: $x \in \mathit{assertion} \Longrightarrow y \in \mathit{assertion} \Longrightarrow x \sqcup y \in \mathit{assertion}$
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*: $x * \top \sqcap x \frown o = x$
assume *B*: $y * \top \sqcap y \frown o = y$
have $(y * \top) \sqcap (x \frown o) \sqcap (y \frown o) = (x \frown o) \sqcap (y * \top) \sqcap (y \frown o)$ **by** (*simp add*: *inf-commute*)
also have $\ldots = (x \frown o) \sqcap ((y * \top) \sqcap (y \frown o))$ **by** (*simp add*: *inf-assoc*)
also have $\ldots = (x \frown o) \sqcap y$ **by** (*simp add*: *B*)
also have $\ldots = y$
apply (*rule order.antisym*, *simp-all*)
apply (*rule-tac* $y = 1$ **in** *order-trans*)
apply *simp*
by (*subst dual-le*, *simp*)
finally have [*simp*]: $(y * \top) \sqcap (x \frown o) \sqcap (y \frown o) = y$ **.**
have $x * \top \sqcap (x \frown o) \sqcap (y \frown o) = x \sqcap (y \frown o)$ **by** (*simp add*: *A*)
also have $\ldots = x$
apply (*rule order.antisym*, *simp-all*)
apply (*rule-tac* $y = 1$ **in** *order-trans*)
apply *simp*
by (*subst dual-le*, *simp*)
finally have [*simp*]: $x * \top \sqcap (x \frown o) \sqcap (y \frown o) = x$ **.**
have $(x \sqcup y) * \top \sqcap (x \sqcup y) \frown o = (x * \top \sqcup y * \top) \sqcap ((x \frown o) \sqcap (y \frown o))$ **by** (*simp add*: *sup-comp dual-sup*)
also have $\ldots = x \sqcup y$ **by** (*simp add*: *inf-sup-distrib inf-assoc* [*THEN sym*])
finally show $(x \sqcup y) * \top \sqcap (x \sqcup y) \frown o = x \sqcup y$ **.**
qed

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

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

lemma [*simp*]: $x \in \mathit{assertion} \Longrightarrow x * (x \frown o) = x$
proof −
assume *A*: $x \in \mathit{assertion}$
have *B*: $x * \top \sqcap (x \frown o) = x$ **by** (*cut-tac A*, *unfold assertion-def*, *simp*)
have $x * x \frown o = (x * \top \sqcap (x \frown o)) * x \frown o$ **by** (*simp add*: *B*)
also have $\ldots = x * \top \sqcap (x \frown o)$ **by** (*cut-tac A*, *simp add*: *inf-comp*)

```
    also have ... = x by (simp add: B)
    finally show ?thesis .
  qed

lemma [simp]: x ∈ assertion ⟹ (x ⌢ o) * x = x ⌢ o
  apply (rule dual-eq)
  by (simp add: dual-comp)


lemma [simp]: ⊥ ∈ assertion
  by (unfold assertion-def, simp)

lemma [simp]: 1 ∈ assertion
  by (unfold assertion-def, simp)
```

3.2 Weakest precondition of true

```
definition
  wpt x = (x * ⊤) ⊓ 1

lemma wpt-is-assertion [simp]: wpt x ∈ 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 = ⊤ 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 ∈ assertion ⟹ wpt x = x
  by (simp add: wpt-def assertion-prop)

lemma wpt-le-assertion: x ∈ assertion ⟹ x * y = y ⟹ wpt y ≤ x
  apply (simp add: wpt-def)
  proof −
    assume A: x ∈ assertion
    assume B: x * y = y
    have y * ⊤ ⊓ 1 = x * (y * ⊤) ⊓ 1 by (simp add: B mult.assoc [THEN sym])
    also have ... ≤ x * ⊤ ⊓ 1
      apply simp
      apply (rule-tac y = x * (y * ⊤) in order-trans)
      apply simp-all
      by (rule le-comp, simp)
```

also have ... = 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 ... = $x * \top \sqcap (y * \top \sqcap 1)$ **by** (*subst inf-assoc*, *simp*)
also have ... = $(x * \top) \sqcap (y * \top) \sqcap 1$ **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*]: $x \leq y \Longrightarrow x \sqcap y = x$
by (*simp add*: *inf-absorb1*)
end

context *mbt-algebra* **begin**

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

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

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

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

lemma *assertion-commute*: $x \in assertion \Longrightarrow y \in conjunctive \Longrightarrow y * x = wpt(y * x) * y$
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*: $x \leq y \Longrightarrow wpt\ x \leq wpt\ y$
apply (*simp add*: *wpt-def*)
apply (*rule-tac* $y = x * \top$ **in** *order-trans*, *simp-all*)
by (*rule le-comp-right*, *simp*)

lemma $a \in conjunctive \Longrightarrow x * a \leq a * y \Longrightarrow (x \frown \omega) * a \leq a * (y \frown \omega)$
apply (*rule omega-least*)
apply (*simp add*: *mult.assoc* [*THEN sym*])
apply (*rule-tac* $y = a * y * y \frown \omega \sqcap a$ **in** *order-trans*)
apply (*simp*)
apply (*rule-tac* $y = x * a * y \frown \omega$ **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*]: $x \leq 1 \Longrightarrow y * x \leq y$
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*]: $x * \bot \leq x$
by (*cut-tac* $x = \bot$ **and** $y = 1$ **and** $z = x$ **in** *le-comp*, *simp-all*)

end

3.3 Monotonic Boolean trasformers algebra with post condition statement

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

lemma *mono-post-fun* [*simp*]: *mono* (*post-fun* (p::-::{*order-bot*,*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* $\Longrightarrow$ ((*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 λx. *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*: $(Inf\ X)\ \widehat{}\ o = (SUP\ x{\in}\ X\ .\ x\ \widehat{}\ o)$
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*: $(Sup\ X)\ \widehat{}\ o = (INF\ x{\in}\ X\ .\ x\ \widehat{}\ o)$
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*: $(\bigsqcap(f\ `\ A))\ *\ z = (INF\ a \in A\ .\ (f\ a)\ *\ z)$
unfolding *Inf-comp*
apply (*subgoal-tac* $((\lambda x{::}'a.\ x\ *\ z)\ `\ f\ `\ A) = ((\lambda a{::}'b.\ f\ a\ *\ z)\ `\ A)$)
by *auto*

lemma *dual-INF*: $(\bigsqcap(f\ `\ A))\ \widehat{}\ o = (SUP\ a \in A\ .\ (f\ a)\ \widehat{}\ o)$
unfolding *Inf-comp dual-Inf*
apply (*subgoal-tac* $(dual\ `\ f\ `\ A) = ((\lambda a{::}'b.\ f\ a\ \widehat{}\ o)\ `\ A)$)
by *auto*

lemma *dual-SUP*: $(\bigsqcup(f\ `\ A))\ \widehat{}\ o = (INF\ a \in A\ .\ (f\ a)\ \widehat{}\ o)$
unfolding *dual-Sup*
apply (*subgoal-tac* $(dual\ `\ f\ `\ A) = ((\lambda a{::}'b.\ f\ a\ \widehat{}\ o)\ `\ A)$)
by *auto*

lemma *Sup-comp*: $(Sup\ X)\ *\ z = (SUP\ x \in X\ .\ (x\ *\ z))$
apply (*rule dual-eq*)
by (*simp add*: *dual-comp dual-Sup dual-SUP INF-comp image-comp*)

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

lemma *Sup-assertion* [*simp*]: $X \subseteq assertion \Longrightarrow Sup\ X \in assertion$
apply (*unfold assertion-def*)
apply *safe*

apply (*rule Sup-least*)
apply *blast*
apply (*simp add*: *Sup-comp dual-Sup Sup-inf*)
apply (*subgoal-tac* $((\lambda y\ .\ y \sqcap \bigsqcap (dual\ `\ X))\ `\ (\lambda x\ .\ x * \top)\ `\ X) = X$)
apply *simp*
proof −
assume *A*: $X \subseteq \{x.\ x \leq 1 \wedge x * \top \sqcap x \frown o = x\}$
have *B* [*simp*]: $!!\ x\ .\ x \in X \Longrightarrow\ x * \top \sqcap (\bigsqcap (dual\ `\ X)) = x$
proof −
fix *x*
assume *C*: $x \in X$
have $x * \top \sqcap \bigsqcap (dual\ `\ X) = x * \top \sqcap (x \frown o \sqcap \bigsqcap (dual\ `\ X))$
apply (*subgoal-tac* $\bigsqcap (dual\ `\ X) = (x \frown o \sqcap \bigsqcap (dual\ `\ X))$, *simp*)
apply (*rule order.antisym*, *simp-all*)
apply (*rule Inf-lower*, *cut-tac C*, *simp*)
done
also have $\ldots = x \sqcap \bigsqcap (dual\ `\ X)$ **by** (*unfold inf-assoc* [*THEN sym*], *cut-tac A*, *cut-tac C*, *auto*)
also have $\ldots = x$
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 $x * \top \sqcap \bigsqcap (dual\ `\ X) = x$ **.**
qed
show $(\lambda y.\ y \sqcap \bigsqcap (dual\ `\ X))\ `\ (\lambda x\ .\ x * \top)\ `\ X = X$
by (*simp add*: *image-comp*)
qed

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

lemma *Sup-less-assertion* [*simp*]: $(!!w\ .\ p\ w \in assertion) \Longrightarrow Sup\text{-}less\ p\ w \in assertion$
by (*unfold Sup-less-def*, *rule Sup-assertion*, *auto*)

theorem *omega-lfp*:
$x \frown \omega * y = lfp\ (\lambda\ z\ .\ (x * z) \sqcap y)$
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* (λ (*t*::'*a*::*mbt-algebra*) . *x* ∗ *t* ⊓ *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*: (∀ *t* . *x* ∗ *t* ⊓ *y* ⊓ *z* ≤ *u* ∗ (*t* ⊓ *z*) ⊓ *v*)
⟹ (*x* ⁀ω) ∗ *y* ⊓ *z* ≤ (*u* ⁀ω) ∗ *v*

lemma
class.mbt-algebra-fusion (*1*::'*a*::*complete-mbt-algebra*) ((∗)) (⊓) (≤) (<) (⊔) *dual dual-star omega star* ⊥ ⊤
apply *unfold-locales*
apply (*cut-tac h* = λ *t* . *t* ⊓ *z* **and** *f* = λ *t* . *x* ∗ *t* ⊓ *y* **and** *g* = λ *t* . *u* ∗ *t* ⊓ *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* ∈ *conjunctive* ⟹ *x* ⁀ω = *wpt* (*x* ⁀ω) ∗ (*x* ⁀∗)
apply (*simp add*: *wpt-def inf-comp*)
apply (*rule order.antisym*)
apply (*cut-tac x* = *x* **and** *y* = *1* **and** *z* = *x* ⁀ω ∗ ⊤ ⊓ *x* ⁀∗ **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* = ⊤ **and** *z* = *x* ⁀∗ **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* ∈ *conjunctive* ⟹ *x* ⁀ω ∈ *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 instroduced 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* = ⊥ **in** *exI*)
  by (*unfold assertion-def*, *simp*)

definition
  assert :: '*a*::*mbt-algebra Assertion* ⇒ '*a* (‹{· - }› [*0*] *1000*) **where**
  {·*p*} = *Rep-Assertion p*

definition
  abs-wpt x = *Abs-Assertion* (*wpt x*)

lemma [*simp*]: {·*p*} ∈ *assertion*
  by (*unfold assert-def*, *cut-tac x* = *p* **in** *Rep-Assertion*, *simp*)

lemma [*simp*]: *abs-wpt* ({·*p*}) = *p*
  apply (*simp add*: *abs-wpt-def*)
  by (*simp add*: *assert-def Rep-Assertion-inverse*)

lemma [*simp*]: *x* ∈ *assertion* ⟹ {·*Abs-Assertion x*} = *x*
  apply (*simp add*: *assert-def*)
  by (*rule Abs-Assertion-inverse*, *simp*)

lemma [*simp*]: *x* ∈ *assertion* ⟹ {·*abs-wpt x*} = *x*
  apply (*simp add*: *abs-wpt-def assert-def*)
  by (*rule Abs-Assertion-inverse*, *simp*)

lemma *assert-injective*: {·*p*} = {·*q*} ⟹ *p* = *q*
  proof −
    assume *A*: {· *p* } = {· *q* }
    have *p* = *abs-wpt* ({·*p*}) **by** *simp*
    also have ... = *q* **by** (*subst A*, *simp*)
    finally show *?thesis* **.**
  qed

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

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

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

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

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

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

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: $(p{::}'a\ Assertion) - q = p \sqcap - q$

instance
proof
fix $x\ y :: 'a\ Assertion$ **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 $x ::'a\ Assertion$ **show** $x \leq x$ **by** (*simp add*: *less-eq-Assertion-def*)
next
fix $x\ y\ z :: 'a\ Assertion$ **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 $x\ y :: 'a\ Assertion$ **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 $x\ y :: 'a\ Assertion$ **show** $x \sqcap y \leq x$
by (*simp add*: *less-eq-Assertion-def inf-Assertion-def*)
fix $x\ y :: 'a\ Assertion$ **show** $x \sqcap y \leq y$
by (*simp add*: *less-eq-Assertion-def inf-Assertion-def*)
next
fix $x\ y\ z :: 'a\ Assertion$ **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** $\bot \leq x$
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** $x \sqcap - x = \bot$
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*) $\in A \Longrightarrow x \leq$ *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* . $x \in A \Longrightarrow x \leq z$) $\Longrightarrow$ *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: $x \in A$ **from** A **show** *Inf* $A \leq x$
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: $\bigwedge x\ .\ x \in A \Longrightarrow z \leq x$ **from** A **show** $z \leq$ *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: $x \in A$ **from** A **show** $x \leq$ *Sup* A
by (*rule Sup1*)
next
fix $z :: 'a$ *Assertion* **fix** A **assume** A: $\bigwedge x\ .\ x \in A \Longrightarrow x \leq z$ **from** A **show** *Sup* $A \leq z$
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* $\{\}$ = $(\bot :: 'a$ *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*: $A \neq \{\} \Longrightarrow \{\cdot Inf\ A\} = Inf\ (assert\ `\ A)$
apply (*unfold assert-Inf*)
apply (*rule antisym*)
apply *simp-all*
apply *safe*
apply (*erule notE*)
apply (*rule-tac y* = $\{\cdot x\}$ **in** *order-trans*)
by (*simp-all add*: *INF-lower*)

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

lemma *assert-Sup-less*: $\{\cdot\ Sup\text{-}less\ p\ w\ \} = Sup\text{-}less\ (assert\ o\ p)\ w$
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 refienment. 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* ⇒ *'a* (‹[· -]› [*0*] *1000*) **where**
[·*p*] = {·*p*} ⌢ *o*

lemma [*simp*]: {·*p*} ∗ ⊤ ⊓ [·*p*] = {·*p*}
apply (*subgoal-tac* {·*p*} ∈ *assertion*)
apply (*subst* (*asm*) *assertion-def*, *simp add*: *assume-def*)
by *simp*

lemma [*simp*]: [·*p*] ∗ *x* ⊔ {·−*p*} ∗ ⊤ = [·*p*] ∗ *x*
by (*simp add*: *assume-def uminus-Assertion-def*)

lemma [*simp*]: {·*p*} ∗ ⊤ ⊔ [·−*p*] ∗ *x* = [·−*p*] ∗ *x*
by (*simp add*: *assume-def uminus-Assertion-def*)

lemma *assert-sup*: {·*p* ⊔ *q*} = {·*p*} ⊔ {·*q*}
by (*simp add*: *sup-Assertion-def*)

lemma *assert-inf*: {·*p* ⊓ *q*} = {·*p*} ⊓ {·*q*}
by (*simp add*: *inf-Assertion-def*)

lemma *assert-neg*: {·−*p*} = *neg-assert* {·*p*}
by (*simp add*: *uminus-Assertion-def*)

lemma *assert-false* [*simp*]: {·⊥} = ⊥
by (*simp add*: *bot-Assertion-def*)

lemma *if-Assertion-assumption*: ({·*p*} ∗ *x*) ⊔ ({·−*p*} ∗ *y*) = ([·*p*] ∗ *x*) ⊓ ([·−*p*] ∗ *y*)
proof −
have ({·*p*} ∗ *x*) ⊔ {·−*p*} ∗ *y* = ({·*p*} ∗ ⊤ ⊓ [·*p*]) ∗ *x* ⊔ ({·−*p*} ∗ ⊤ ⊓ [·−*p*]) ∗ *y* **by** *simp*
also have ... = ({·*p*} ∗ ⊤ ⊓ ([·*p*] ∗ *x*)) ⊔ ({·−*p*} ∗ ⊤ ⊓ ([·−*p*] ∗ *y*)) **by** (*unfold inf-comp*, *simp*)
also have ... = (({·*p*} ∗ ⊤ ⊓ ([·*p*] ∗ *x*)) ⊔ ({·−*p*} ∗ ⊤)) ⊓ (({·*p*} ∗ ⊤ ⊓ ([·*p*] ∗ *x*)) ⊔ ([·−*p*] ∗ *y*)) **by** (*simp add*: *sup-inf-distrib*)
also have ... = (({·*p*} ∗ ⊤ ⊔ ({·−*p*} ∗ ⊤)) ⊓ (([·*p*] ∗ *x*))) ⊓ (([·−*p*] ∗ *y*) ⊓ (([·*p*] ∗ *x*) ⊔ ([·−*p*] ∗ *y*)))
by (*simp add*: *sup-inf-distrib2*)
also have ... = ([·*p*] ∗ *x*) ⊓ ([·−*p*] ∗ *y*) ⊓ (([·*p*] ∗ *x*) ⊔ ([·−*p*] ∗ *y*))
apply (*simp add*: *sup-comp* [*THEN sym*])
by (*simp add*: *assert-sup* [*THEN sym*] *inf-assoc*)
also have ... = ([·*p*] ∗ *x*) ⊓ ([·−*p*] ∗ *y*)
by (*rule antisym*, *simp-all add*: *inf-assoc*)
finally show *?thesis* **.**
qed

definition
wp x p = abs-wpt (x ∗ {·p})

lemma *wp-assume*: *wp [·p] q = −p ⊔ 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 ∈ conjunctive ⟹ y ∗ {·p} = {· wp y p } ∗ y*
apply (*simp add*: *wp-def abs-wpt-def*)
by (*rule assertion-commute*, *simp-all*)

lemma *wp-assert*: *wp {·p} q = p ⊓ 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 ∗ {· xa } ∗ ⊤* **in** *order-trans*, *simp-all*)
apply (*rule le-comp-right*)
by (*rule le-comp*, *simp*)

lemma *wp-mono2*: *p ≤ q ⟹ wp x p ≤ 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 ∗ {· xa } ∗ ⊤* **in** *order-trans*, *simp-all*)
apply (*rule le-comp-right*)
by (*rule le-comp-right*, *simp*)

lemma *wp-fun-mono2*: *x ≤ y ⟹ wp x p ≤ 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 wpt-comp-2* [*THEN sym*] *mult.assoc*, *simp*)

lemma *wp-choice*: *wp (x ⊓ y) = wp x ⊓ 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* ^ *ω*) *p* = *wp x* (*wp* (*x* ^ *ω*) *p*) ⊓ *p*
apply (*subst omega-fix*)
by (*simp add*: *wp-choice wp-comp*)

lemma *wp-omega-least*: (*wp x r*) ⊓ *p* ≤ *r* ⟹ *wp* (*x* ^ *ω*) *p* ≤ *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*} ∗ ⊤ ⊓ *1* **in** *order-trans*)
apply *simp*
apply (*rule-tac y* = *x* ^ *ω* ∗ {· *p* } ∗ ⊤ **in** *order-trans*, *simp*)
apply (*simp add*: *mult.assoc*)
apply (*rule omega-least*)
apply (*drule-tac z* = ⊤ **in** *le-comp-right*)
apply (*simp add*: *inf-comp mult.assoc* [*THEN sym*])
by (*simp add*: *assertion-prop*)

lemma *Assertion-wp*: {·*wp x p*} = (*x* ∗ {·*p*} ∗ ⊤) ⊓ *1*
apply (*simp add*: *wp-def abs-wpt-def*)
by (*simp add*: *wpt-def*)

definition
hoare p S q = (*p* ≤ *wp S q*)

definition
grd x = − (*wp x* ⊥)

lemma *grd-comp*: [·*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*: {·*p*} ∗ [·*p*] = {· *p*}
by (*simp add*: *assume-def*)

lemma *dual-assume*: [·*p*] ^ *o* = {·*p*}
by (*simp add*: *assume-def*)

lemma *assume-prop*: ([·*p*] ∗ ⊥) ⊔ *1* = [·*p*]
by (*simp add*: *assume-def dual-assertion-prop*)

An alternative definition of a Hoare triple

definition *hoare1 p S q* = ([· *p*] ∗ *S* ∗ [· −*q*] = ⊤)

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*: [· *p*] ∗ *S* ∗ [· − *q*] = ⊤
have {·*p*} ≤ {·*p*} ∗ ⊤ **by** *simp*
also have ... ≤ {·*p*} ∗ ⊤ ∗ ⊥ **by** (*unfold mult.assoc*, *simp*)
also have ... = {·*p*} ∗ [· *p*] ∗ *S* ∗ [· − *q*] ∗ ⊥ **by** (*subst A* [*THEN sym*], *simp add*: *mult.assoc*)
also have ... = {·*p*} ∗ *S* ∗ [· − *q*] ∗ ⊥ **by** (*simp add*: *assert-assume*)
also have ... ≤ {·*p*} ∗ *S* ∗ {· *q* } ∗ ⊤
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 ... ≤ *S* ∗ {· *q* } ∗ ⊤ **by** (*simp add*: *mult.assoc*)
finally show {·*p*} ≤ *S* ∗ {· *q* } ∗ ⊤ **.**
next
assume *A*: {· *p* } ≤ *S* ∗ {· *q* } ∗ ⊤
have ⊤ = ((*S* ∗ {·*q*}) ⌢ *o*) ∗ ⊥ ⊔ *S* ∗ {·*q*} ∗ ⊤ **by** *simp*
also have . . . ≤ [·*p*] ∗ ⊥ ⊔ *S* ∗ {·*q*} ∗ ⊤
apply (*simp del*: *dual-neg-top*)
apply (*rule-tac y* = [·*p*] ∗ ⊥ **in** *order-trans*, *simp-all*)
apply (*subst dual-le*)
apply (*simp add*: *dual-comp dual-assume*)
apply (*cut-tac x* = {·*p*} **and** *y* = *S* ∗ {·*q*} ∗ ⊤ **and** *z* = ⊤ **in** *le-comp-right*)
apply (*rule A*)
by (*simp add*: *mult.assoc*)
also have . . . = [·*p*] ∗ *S* ∗ ({·*q*} ∗ ⊤)
apply (*subst* (*2*) *assume-prop* [*THEN sym*])
by (*simp-all add*: *sup-comp mult.assoc*)
also have . . . ≤ [·*p*] ∗ *S* ∗ ({·*q*} ∗ ⊤ ⊔ *1*)
by (*rule le-comp*, *simp*)
also have . . . = [·*p*] ∗ *S* ∗ [·−*q*]
apply (*simp add*: *assume-def uminus-Assertion-def*)
by (*simp add*: *neg-assert-def dual-inf dual-comp*)
finally show [·*p*] ∗ *S* ∗ [· − *q*] = ⊤
by (*rule-tac antisym*, *simp-all*)
qed

lemma *hoare-choice*: *hoare p* (*x* ⊓ *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* ⇒ *'a* ⇒ *'a* ⇒ *'a* (‹(*If* (-)/ *then* (-)/ *else* (-))› [*0*, *0*, *10*] *10*) **where**
if-stm b x y = (([· *b*] ∗ *x*) ⊓ ([· −*b*] ∗ *y*))

lemma *if-assertion*: (*If p then x else y*) = {·*p*} ∗ *x* ⊔ {· −*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* ⊓ *b*) *x q* ∧ *hoare* (*p* ⊓ −*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* = (∃ *r* . (*hoare p x r*) ∧ (*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*}) ≤ *S*)
apply (*simp add*: *hoare-def less-eq-Assertion-def Assertion-wp*)
proof
assume *A*: {·*p*} ≤ *S* ∗ {·*q*} ∗ ⊤
have {·*p*} ∗ *post* {·*q*} = ({·*p*} ∗ ⊤ ⊓ *1*) ∗ *post* {·*q*} **by** (*simp add*: *assertion-prop*)
also have ... = {·*p*} ∗ ⊤ ⊓ *post* {·*q*} **by** (*simp add*: *inf-comp*)
also have ... ≤ *S* ∗ {·*q*} ∗ ⊤ ⊓ *post* {·*q*} **apply** *simp*
apply (*rule-tac y* = {·*p*} ∗ ⊤ **in** *order-trans*, *simp-all*)
apply (*cut-tac x* = {·*p*} **and** *y* = *S* ∗ {·*q*} ∗ ⊤ **and** *z* = ⊤ **in** *le-comp-right*)
by (*rule A*, *simp*)
also have ... ≤ *S* **by** (*simp add*: *post-2*)
finally show {·*p*} ∗ *post* {·*q*} ≤ *S*.
next
assume *A*: {·*p*} ∗ *post* {·*q*} ≤ *S*
have {·*p*} = {·*p*} ∗ ⊤ ⊓ *1* **by** (*simp add*: *assertion-prop*)
also have ... = {·*p*} ∗ ((*post* {·*q*}) ∗ {·*q*} ∗ ⊤) ⊓ *1* **by** (*simp add*: *post-1*)
also have ... ≤ {·*p*} ∗ ((*post* {·*q*}) ∗ {·*q*} ∗ ⊤) **by** *simp*
also have ... ≤ *S* ∗ {·*q*} ∗ ⊤
apply (*cut-tac x* = {·*p*} ∗ *post* {·*q*} **and** *y* = *S* **and** *z* = {·*q*} ∗ ⊤ **in** *le-comp-right*)
apply (*simp add*: *A*)
by (*simp add*: *mult.assoc*)
finally show {·*p*} ≤ *S* ∗ {·*q*} ∗ ⊤ .
qed

theorem *hoare-fixpoint-mbt*:
F x = *x*
⟹ (!! (*w*::*'a*::*well-founded*) *f* . (⋀*v*. *v* < *w* ⟹ *hoare* (*p v*) *f q*) ⟹ *hoare* (*p w*) (*F f*) *q*)

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

lemma *hoare-Sup*: *hoare* (*Sup P*) *x q* = (∀ *p* ∈ *P* . *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*
$\Longrightarrow$ (!! *w f* . *hoare* (*Sup-less p w*) *f q* $\Longrightarrow$ *hoare* (*p w*) (*F f*) *q*)
$\Longrightarrow$ *hoare* (*Sup* (*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
while:: *'a*::*mbt-algebra Assertion* ⇒ *'a* ⇒ *'a* (‹(*While* (-)/ *do* (-))› [*0*, *10*] *10*)
where
while p x = ([· *p*] ∗ *x*) ^ω ∗ [· −*p*]

lemma *while-false*: (*While* ⊥ *do x*) = *1*
apply (*unfold while-def*)
apply (*subst omega-fix*)
by (*simp-all add*: *assume-def*)

lemma *while-true*: (*While* ⊤ *do 1*) = ⊥
apply (*unfold while-def*)
by (*rule antisym*, *simp-all add*: *assume-def*)

lemma *hoare-wp* [*simp*]: *hoare* (*wp x q*) *x q*
by (*simp add*: *hoare-def*)

lemma *hoare-comp-wp*: *hoare p* (*x* ∗ *y*) *q* = *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*:
$(\forall\ (w::'b::well\text{-}founded)\ r\ .\ (\forall\ v\ .\ v < w \longrightarrow p\ v \leq r) \longrightarrow hoare\ ((p\ w) \sqcap b)\ x\ r) \Longrightarrow$
$(\forall\ u\ .\ p\ u \leq q) \Longrightarrow hoare\ \ (p\ w)\ (While\ b\ do\ x)\ (q \sqcap -b)$
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*:
$(\forall\ w::'b::well\text{-}founded\ .\ hoare\ ((p\ w) \sqcap b)\ x\ (Sup\text{-}less\ p\ w)) \Longrightarrow$
$hoare\ \ (Sup\ (range\ p))\ (While\ b\ do\ x)\ ((Sup\ (range\ p)) \sqcap -b)$
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* $y = wp\ x\ (Sup\text{-}less\ p\ w)$ **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* ⟹ *datarefin* ({·*p*} ∗ *S*) *S1 D D1* ⟹ *hoare* (*wp D p*) *S1* (*wp D1 q*)
apply (*simp add*: *hoare-def datarefin-def*)
apply (*rule-tac y* = *wp* (*D* ∗ {·*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* ∈ *conjunctive* ⟹ *y* ∈ *conjunctive* ⟹ *x* ⊓ *y* ∈ *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* ∈ *disjunctive* ⟹ *y* ∈ *disjunctive* ⟹ *x* ⊔ *y* ∈ *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-conjuncive* [*simp*]: [·*p*] ∈ *conjunctive*
by (*simp add*: *assume-def dual-disjunctive assertion-disjunctive*)

lemma *assumption-disjuncive* [*simp*]: [·*p*] ∈ *disjunctive*
by (*simp add*: *assume-def dual-conjunctive assertion-conjunctive*)

lemma *if-pres-conj*: *x* ∈ *conjunctive* ⟹ *y* ∈ *conjunctive* ⟹ (*If p then x else y*) ∈ *conjunctive*
apply (*unfold if-stm-def*)
by (*simp add*: *inf-pres-conj comp-pres-conj*)

lemma *if-pres-disj*: *x* ∈ *disjunctive* ⟹ *y* ∈ *disjunctive* ⟹ (*If p then x else y*) ∈ *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*)) = (({· *p*} ∗ *x*)^⊗ ∗ {· −*p* })
apply (*simp add*: *while-def*)

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

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