Aggregation Algebras

Walter Guttmann

September 16, 2018

Abstract

We develop algebras for aggregation and minimisation for weight matrices and for edge weights in graphs. We verify the correctness of Prim’s and Kruskal’s minimum spanning tree algorithms based on these algebras. We also show numerous instances of these algebras based on linearly ordered commutative semigroups.

Contents

1 Overview 2

2 Big Sum over Finite Sets in Abelian Semigroups 3
   2.1 Generic Abelian semigroup operation over a set ............ 3
   2.2 Generalized summation over a set .......................... 13
     2.2.1 Properties in more restricted classes of structures ... 15

3 Algebras for Aggregation and Minimisation 18

4 Matrix Algebras for Aggregation and Minimisation 22
   4.1 Aggregation Orders and Finite Sums ........................ 22
   4.2 Matrix Aggregation ......................................... 26
   4.3 Aggregation Lattices ........................................ 28
   4.4 Matrix Minimisation ......................................... 33
   4.5 Linear Aggregation Lattices ................................ 35

5 Algebras for Aggregation and Minimisation with a Linear
   Order 44
   5.1 Linearly Ordered Commutative Semigroups ................. 44
   5.2 Linearly Ordered Commutative Monoids .................... 48
   5.3 Linearly Ordered Commutative Monoids with a Least Element 51
   5.4 Linearly Ordered Commutative Monoids with a Greatest El-
      ement ....................................................... 55
   5.5 Linearly Ordered Commutative Monoids with a Least Ele-
      ment and a Greatest Element .............................. 60
1 Overview

This document describes the following seven theory files:

* Big sums over semigroups generalises parts of Isabelle/HOL’s theory of finite summation Groups_Big.thy from commutative monoids to commutative semigroups with a unit element only on the image of the semigroup operation.

* Aggregation Algebras introduces s-algebras, m-algebras and m-Kleene-algebras with operations for aggregating the elements of a weight matrix and finding the edge with minimal weight.

* Matrix Aggregation Algebras introduces aggregation orders, aggregation lattices and linear aggregation lattices. Matrices over these structures form s-algebras and m-algebras.

* Linear Aggregation Algebras shows numerous instances based on linearly ordered commutative semigroups. They include aggregations used for the minimum weight spanning tree problem and for the minimum bottleneck spanning tree problem, as well as arbitrary t-norms and t-conorms.

* Hoare Logic is a light-weight modification of Isabelle/HOL’s theory Hoare/Hoare_Logic.thy for total-correctness proofs.

* Hoare Logic Examples gives a few simple total-correctness proof examples.

* Minimum Spanning Trees proves total correctness of Kruskal’s and Prim’s algorithms based on m-Kleene-algebras using Hoare logic.

The development is based on Stone-Kleene relation algebras [3, 2]. The algebras for aggregation and minimisation, their application to weighted graphs and the verification of Prim’s and Kruskal’s minimum spanning tree algorithms, and various instances of aggregation are described in [1, 4, 5]. Related work is discussed in these papers.
2  Big Sum over Finite Sets in Abelian Semigroups

theory Semigroups-Big
  imports HOL.Power
begin

This theory is based on Isabelle/HOL’s Groups-Big.thy written by T. Nipkow, L. C. Paulson, M. Wenzel and J. Avigad. We have generalised a selection of its results from Abelian monoids to Abelian semigroups with an element that is a unit on the image of the semigroup operation.

2.1  Generic Abelian semigroup operation over a set

locale abel-semigroup-set = abel-semigroup +
  fixes z :: 'a (1)
  assumes z-neutral [simp]: x * y * 1 = x * y
  assumes z-idem [simp]: 1 * 1 = 1
begin

interpretation comp-fun-commute f
  by standard (simp add: fun-eq-iff left-commute)

interpretation comp?: comp-fun-commute f ◦ g
  by (fact comp-comp-fun-commute)

definition F :: ('b ⇒ 'a) ⇒ 'b set ⇒ 'a
  where eq-fold: F g A = Finite-Set.fold (f ◦ g) 1 A

lemma infinite [simp]: ¬ finite A ⇒ F g A = 1
  by (simp add: eq-fold)

lemma empty [simp]: F g {} = 1
  by (simp add: eq-fold)

lemma insert [simp]: finite A ⇒ x ∉ A ⇒ F g (insert x A) = g x * F g A
  by (simp add: eq-fold)

lemma remove:
  assumes finite A and x ∈ A
  shows F g A = g x * F g (A − {x})
proof −
  from ⟨x ∈ A⟩ obtain B where B: A = insert x B and x ∉ B
    by (auto dest: mk-disjoint-insert)
  moreover from ⟨finite A⟩ ⟨finite B⟩ have finite B by simp
  ultimately show ?thesis by simp
qed

lemma insert-remove: finite A ⇒ F g (insert x A) = g x * F g (A − {x})
lemma insert-if: finite A \implies F g \ (\text{insert} \ x \ A) = (\text{if} \ x \in A \ then \ F g \ A \ else \ g \ x \star F g \ A)
  by (cases x \in A) (simp-all add: insert-absorb)

lemma neutral: \forall x \in A. \ g x = 1 \implies F g \ A = 1
  by (induct A rule: infinite-finite-induct) simp-all

lemma neutral-const [simp]: F (\lambda x. 1) \ A = 1
  by (simp add: neutral)

proof
  have \forall f b B. \ F f \ (\text{insert} \ b :: b) B \star 1 = F f \ (\text{insert} b B) \vee \text{infinite} B
    using insert-remove by fastforce
  then show ?thesis
    by (metis (no-types) all-not-in-conv empty z-idem infinite insert-if)
qed

lemma one-F [simp]: 1 \star F g \ A = F g \ A
  using F-one commute by auto

lemma F-g-one [simp]: F (\lambda x. g x \star 1) \ A = F g \ A
  apply (induct A rule: infinite-finite-induct)
  apply simp
  by (metis one-F assoc insert-left-commute)

lemma union-inter:
  assumes finite A and finite B
  shows F g \ ((A \cup B) \star F g \ (A \cap B)) = F g \ A \star F g \ B
    — The reversed orientation looks more natural, but LOOPS as a simprule!
    using assms
  proof (induct A)
    case empty
    then show ?case by simp
  next
    case (insert x A)
    then show ?case
      by (auto simp: insert-absorb Int-insert-left commute [of - g x] assoc left-commute)
  qed

corollary union-inter-neutral:
  assumes finite A and finite B
    and \forall x \in A \cap B. \ g x = 1
  shows F g \ (A \cup B) = F g \ A \star F g \ B
  using assms by (simp add: union-inter [symmetric] neutral)
corollary union-disjoint:
assumes finite A and finite B
assumes \( A \cap B = \{ \} \)
shows \( F g (A \cup B) = F g A \ast F g B \)
using assms by (simp add: union-inter-neutral)

lemma union-diff2:
assumes finite A and finite B
shows \( F g (A \cup B) = F g (A - B) \ast F g (B - A) \ast F g (A \cap B) \)
proof –
have \( A \cup B = A - B \cup (B - A) \cup A \cap B \)
  by auto
with assms show \( \text{thesis} \)
  by simp (subst union-disjoint, auto)+
qed

lemma subset-diff:
assumes \( B \subseteq A \) and finite A
shows \( F g A = F g (A - B) \ast F g B \)
proof –
from assms have finite \((A - B)\) by auto
moreover from assms have finite B by (rule finite-subset)
moreover from assms have \((A - B) \cap B = \{ \} \) by auto
ultimately have \( F g (A - B \cup B) = F g (A - B) \ast F g B \) by (rule union-disjoint)
moreover from assms have \( A \cup B = A \) by auto
ultimately show \( \text{thesis} \) by simp
qed

lemma setdiff-irrelevant:
assumes finite A
shows \( F g (A - \{ x . \ g x = z \}) = F g A \)
using assms by (induct A) (simp-all add: insert-Diff-if)

lemma not-neutral-contains-not-neutral:
assumes \( F g A \neq 1 \)
obtains a where \( a \in A \) and \( g a \neq 1 \)
proof –
from assms have \( \exists a \in A . \ g a \neq 1 \)
proof (induct A rule: infinite-finite-induct)
case infinite
  then show \( \text{?case} \) by simp
next
case empty
  then show \( \text{?case} \) by simp
next
case (insert a A)
  then show \( \text{?case} \) by fastforce
qed
with that show thesis by blast
qed

lemma reindex:
assumes inj-on h A
shows F g (h ' A) = F (g o h) A
proof (cases finite A)
case True
with assms show ?thesis
by (simp add: eq-fold fold-image comp-assoc)
next
case False
with assms have ¬ finite (h ' A) by (blast dest: finite-imageD)
with False show ?thesis by simp
qed

lemma cong [fundef-cong]:
assumes A = B
assumes g-h: \( \forall x. x \in B \implies g x = h x \)
shows F g A = F h B
using g-h unfolding (A = B)
by (rule cong) (use assms in ⟨simp-all add: simp-implies-def⟩)

lemma strong-cong [cong]:
assumes A = B
\( \forall x. x \in B \implies g (l x) = h x \)
shows F (λx. g x) A = F (λx. h x) B
by (rule cong) (use assms in ⟨simp-all add: simp-implies-def⟩)

lemma reindex-cong:
assumes inj-on l B
assumes A = l ' B
assumes \( \forall x. x \in B \implies g (l x) = h x \)
shows F g A = F h B
using assms by (simp add: reindex)

lemma UNION-disjoint:
assumes finite I and \( \forall i \in I. \) finite (A i)
and \( \forall i \in I, \forall j \in I. i \neq j \implies A i \cap A j = {} \)
shows F g (UNION I A) = F (λx. F g (A x)) I
apply (insert assms)
apply (induct rule: finite-induct)
apply simp
apply atomize
apply (subgoal-tac \( \forall i \in Fa. x \neq i \))
prefer 2 apply blast
apply (subgoal-tac A x \∩ UNION Fa A = {}) prefer 2 apply blast
apply (simp add: union-disjoint)
done

lemma Union-disjoint:
assumes \( \forall A \in C. \) finite \( A \) \( \forall B \in C. \) \( A \neq B \) \( \longrightarrow \) \( A \cap B = \{\} \)
shows \( F g (\bigcup C) = (F \circ F) g C \)
proof (cases finite \( C \))
case True
from UNION-disjoint [OF this assms] show ?thesis by simp
next
case False
then show ?thesis by (auto dest: finite-UnionD intro: infinite)
qed

lemma distrib: \( F (\lambda x. g x \ast h x) \) \( A \) = \( F g A \ast F h A \)
by (induct \( A \) rule: infinite-finite-induct) (simp-all add: assoc commute left-commute)

lemma Sigma:
finite \( A \) \( \Longrightarrow \) \( \forall x \in A. \) finite \( (B x) \) \( \Longrightarrow \) \( F (\lambda x. F (g x) (B x)) A = F \) (case-prod \( g \) ) (SIGMA \( x:A. \) \( B x \) )
apply (subst Sigma-def)
apply (subst UNION-disjoint)
apply assumption
apply simp
apply blast
apply (rule cong)
apply rule
apply (simp add: fun-eq-iff)
apply (subst UNION-disjoint)
apply simp
apply simp
apply blast
apply (simp add: comp-def)
done

lemma related:
assumes Re: \( R \ 1 \ 1 \)
and Rop: \( \forall x1 y1 x2 y2. \) \( R x1 x2 \land R y1 y2 \longrightarrow R (x1 \ast y1) (x2 \ast y2) \)
and fin: finite \( S \)
and R-h-g: \( \forall x \in S. \) \( R (h x) (g x) \)
shows \( R (F h S) (F g S) \)
using fin by (rule finite-subset-induct) (use assms in auto)

lemma mono-neutral-cong-left:
assumes finite \( T \)
and \( S \subseteq T \)
and \( \forall i \in T - S. \) \( h i = 1 \)
and \( \forall x. x \in S \longrightarrow g x = h x \)
shows \( F g S = F h T \)
proof
have eq: \(T = S \cup (T - S)\) using \(S \subseteq T\) by blast
have d: \(S \cap (T - S) = \{\}\) using \(S \subseteq T\) by blast
from (finite T) \(S \subseteq T\) have f: finite S finite \((T - S)\)
  by (auto intro: finite-subset)
show ?thesis using assms(4)
  by (simp add: union-disjoint [OF f d], unfolded eq [symmetric] neutral [OF assms(3)])
qed

lemma mono-neutral-cong-right:
  \(\text{finite } T \Longrightarrow S \subseteq T \Longrightarrow \forall i \in T - S. \ g i = 1 \Longrightarrow (\forall x. x \in S \Longrightarrow g x = h x)\)
  \(\Longrightarrow F \ g \ T = F \ h \ S\)
  by (auto intro!: mono-neutral-cong-left [symmetric])

lemma mono-neutral-left: finite T \(\Longrightarrow S \subseteq T \Longrightarrow \forall i \in T - S. \ g i = 1 \Longrightarrow F \ g \ S = F \ g \ T\)
  by (blast intro: mono-neutral-cong-left)

lemma mono-neutral-right: finite T \(\Longrightarrow S \subseteq T \Longrightarrow \forall i \in T - S. \ g i = 1 \Longrightarrow F \ g \ T = F \ g \ S\)
  by (blast intro!: mono-neutral-left [symmetric])

lemma mono-neutral-cong:
  assumes \([\text{simp}]: \text{finite } T \text{ finite } S\)
  and \(\ast:\ \forall i \in T - S \Longrightarrow h i = 1 \ \forall i \in S - T \Longrightarrow g i = 1\)
  and gh: \(\forall x. x \in S \cap T \Longrightarrow g x = h x\)
  shows \(F \ g \ S = F \ h \ T\)
proof
  have F g S = F g \((S \cap T)\)
    by (rule mono-neutral-right)(auto intro: \(\ast\))
  also have \(\ldots = F \ h \ ((S \cap T)\) using refl gh by (rule cong)
  also have \(\ldots = F \ h \ T\)
    by (rule mono-neutral-left)(auto intro: \(\ast\))
  finally show ?thesis .
qed

lemma reindex-bij-betw: bij-betw h S T \(\Longrightarrow F (\lambda x. \ g \ (h \ x)) \ S = F \ g \ T\)
  by (auto simp: bij-betw-def reindex)

lemma reindex-bij-witness:
  assumes witness:
    \(\forall a. \ a \in S \Longrightarrow i (j a) = a\)
    \(\forall a. \ a \in S \Longrightarrow j a \in T\)
    \(\forall b. \ b \in T \Longrightarrow j (i b) = b\)
    \(\forall b. \ b \in T \Longrightarrow i b \in S\)
  assumes eq:
    \(\forall a. \ a \in S \Longrightarrow h (j a) = g a\)
shows \( F \circ g \circ S = F \circ h \circ T \)

proof

have bij-betw j S T
  using bij-betw-byWitness[where A=S and f=j and f'=i and A'=T]
witness by auto

moreover have \( F \circ g \circ S = F \left( \lambda x. h \circ (j \circ x) \right) \)
  by (intro cong) (auto simp: eq)

ultimately show ?thesis
  by (simp add: reindex-bij-betw)

qed

lemma reindex-bij-betw-not-neutral:
  assumes fin: finite S' finite T'
  assumes bij: bij-betw h \((S - S') \oplus (T - T')\)
  assumes nn:
    \( \forall a \in S' \exists b \in T' \. \. g \circ (h \circ a) = z \)
  shows \( F \left( \lambda x. g \circ (h \circ x) \right) \circ S = F \circ g \circ T \)

proof

  have \( \sim F \left( \lambda x. g \circ (h \circ x) \right) \circ S = F \circ g \circ T \)
    using nn

  also have \( \sim F \circ g \circ T \)
    using \( \sim \)

  finally show ?thesis

next

  case False

  then show ?thesis by simp

qed

lemma reindex-nontrivial:
  assumes \( \sim \)
  shows \( F \circ g \circ A = F \circ (g \circ h) \circ A \)

proof

  subst reindex-bij-betw-not-neutral [symmetric]

  show bij-betw h \((A - \\{x \in A. (g \circ h) \circ x = 1\}) \oplus (h \circ A - h \circ \{x \in A. (g \circ h) \circ x = 1\})\)
    using \( \sim \)

  qed

lemma reindex-bij-witness-not-neutral:
  assumes fin: finite S' finite T'

qed
assumes witness:
\[ \wedge a. a \in S - S' \Rightarrow i (j a) = a \]
\[ \wedge a. a \in S - S' \Rightarrow j a \in T - T' \]
\[ \wedge b. b \in T - T' \Rightarrow j (i b) = b \]
\[ \wedge b. b \in T - T' \Rightarrow i b \in S - S' \]
assumes \( nn \):
\[ \wedge a. a \in S' \Rightarrow g a = z \]
\[ \wedge b. b \in T' \Rightarrow h b = z \]
assumes \( eq \):
\[ \wedge a. a \in S \Rightarrow h (j a) = g a \]

\[ \text{shows } F g S = F h T \]

\[ \text{proof } - \]
\[ \text{have bij: bij-betw } j (S - (S' \cap S)) (T - (T' \cap T)) \]
\[ \text{using witness by (intro bij-betw-byWitness[where } f'=i]) \text{ auto} \]
\[ \text{have } F\text{-eq: } F g S = F (\lambda x. h (j x)) S \]
\[ \text{by (intro cong) (auto simp: eq)} \]
\[ \text{show } \text{thesis} \]
\[ \text{using } F\text{-eq } \text{unfolding fin nn eq} \]
\[ \text{by (intro reindex-bij-betw-not-neutral[OF - - bij]) auto} \]
\[ \text{qed} \]

\[ \text{lemma delta-remove:} \]
\[ \text{assumes } fS: \text{finite } S \]
\[ \text{shows } F (\lambda k. \text{if } k = a \text{ then } b k \text{ else } c k) S = (\text{if } a \in S \text{ then } b a \ast F c (S - \{a\}) \text{ else } F c (S - \{a\})) \]

\[ \text{proof } - \]
\[ \text{let } ?f = (\lambda k. \text{if } k = a \text{ then } b k \text{ else } c k) \]
\[ \text{show } \text{thesis} \]
\[ \text{proof (cases } a \in S) \]
\[ \text{case } \text{False} \]
\[ \text{then have } \forall k \in S. \text{ if } k = c k \text{ by simp} \]
\[ \text{with } \text{False} \text{ show } \text{thesis by simp} \]
\[ \text{next} \]
\[ \text{case } \text{True} \]
\[ \text{let } ?A = S - \{a\} \]
\[ \text{let } ?B = \{a\} \]
\[ \text{from } \text{True} \text{ have eq: } S = ?A \cup ?B \text{ by blast} \]
\[ \text{have dj: } ?A \cap ?B = \{\} \text{ by simp} \]
\[ \text{from } fS \text{ have } fAB: \text{finite } ?A \text{ finite } ?B \text{ by auto} \]
\[ \text{have } F \text{ if } S = F \text{ if } A \ast F \text{ if } B \]
\[ \text{using } \text{union-disjoint } [\text{OF } fAB \text{ dj, of if, unfolded eq [symmetric]}] \text{ by simp} \]
\[ \text{with } \text{True} \text{ show } \text{thesis} \]
\[ \text{using } \text{abel-semigroup-set.remove } \text{abel-semigroup-set-axioms } fS \text{ by fastforce} \]
\[ \text{qed} \]

\[ \text{lemma delta [simp]:} \]
\[ \text{assumes } fS: \text{finite } S \]
\[ \text{shows } F (\lambda k. \text{if } k = a \text{ then } b k \text{ else } 1) S = (\text{if } a \in S \text{ then } b a \ast 1 \text{ else } 1) \]
by (simp add: delta-remove [OF assms])

lemma delta′ [simp]:
assumes fin: finite S
shows F (λk. if a = k then b k else 1) S = (if a ∈ S then b a * 1 else 1)
using delta [OF fin, of a b, symmetric] by (auto intro: cong)

lemma If-cases:
fixes P :: ′b ⇒ bool and g h :: ′b ⇒ ′a
assumes fin: finite A
shows F (λx. if P x then h x else g x) A = F h (A ∩ {x. P x}) ∗ F g (A ∩ − {x. P x})
proof
  have a: A = A ∩ {x. P x} ∪ A ∩ − {x. P x} (A ∩ {x. P x}) ∩ (A ∩ − {x. P x}) = {} by blast
  from fin have f: finite (A ∩ {x. P x}) finite (A ∩ − {x. P x}) by auto
  let ?g = λx. if P x then h x else g x
  from union-disjoint [OF f a (2), of ?g] a(1) show ?thesis by (subst (1 2) cong) simp-all qed

lemma cartesian-product: F (λx. F (g x) B) A = F (case-prod g) (A × B)
apply (rule sym)
apply (cases finite A)
apply (cases finite B)
apply (simp add: Sigma)
apply simp
apply simp
apply simp
apply (auto intro: infinite dest: finite-cartesian-productD2)
apply (cases B = {1})
apply (auto intro: infinite dest: finite-cartesian-productD1)
done

lemma inter-restrict:
assumes finite A
shows F g (A ∩ B) = F (λx. if x ∈ B then g x else 1) A
proof
  let ?g = λx. if x ∈ A ∩ B then g x else 1
  have ∀i ∈ A − A ∩ B. (if i ∈ A ∩ B then g i else 1) = 1 by simp
  moreover have A ∩ B ⊆ A by blast
  ultimately have F ?g (A ∩ B) = F ?g A
    using finite A by (intro mono-neutral-left) auto
  then show ?thesis by simp qed

lemma inter-filter:
fine A ⇒ F g {x ∈ A. P x} = F (λx. if P x then g x else 1) A

11
lemma Union-comp:
  assumes \( \forall A \in B. \text{finite } A \)
  and \( \bigwedge A1 A2 x. A1 \in B \implies A2 \in B \implies A1 \neq A2 \implies x \in A1 \implies x \in A2 \)
  \( \implies g x = 1 \)
  shows \( F g (\bigcup B) = (F \circ F) g B \)
  using assms
proof (induct B rule: infinite-finite-induct)
  case (infinite A)
  then have \( \neg \text{finite } (\bigcup A) \) by (blast dest: finite-UnionD)
  with \( \text{infinite} \) show \(?case\) by simp
next
  case empty
  then show \(?case\) by simp
next
  case (insert A B)
  then have \( \text{finite } A \) \( \text{finite } B \) \( \text{finite } (\bigcup B) \) \( A /\notin B \)
  and \( H: F g (\bigcup B) = (F \circ F) g B \) by auto
  then have \( F g (A \cup \bigcup B) = F g A * F g (\bigcup B) \)
    by (simp add: union-inter-neutral)
  with \( \langle \text{finite } B \rangle \langle A /\notin B \rangle \) show \(?case\)
    by (simp add: \( H \))
qed

lemma swap: \( F (\lambda i. F (g i) B) A = F (\lambda j. F (\lambda i. g i j) A) B \)
unfolding cartesian-product
by (rule reindex-bij-witness \[where \ i = \lambda (i, j). (j, i) \text{ and } j = \lambda (i, j). (j, i)\])
auto

lemma swap-restrict:
finite A \( \implies \) finite B \( \implies \)
\( F (\lambda x. F (g x) \{y. y \in B \land R x y\}) A = F (\lambda y. F (\lambda x. g x y) \{x. x \in A \land R x y\}) B \)
by (simp add: inter-filter) (rule swap)

lemma Plus:
fixes A :: 'b set and B :: 'c set
assumes fin: \( \text{finite } A \) \( \text{finite } B \)
shows \( F g (A <+> B) = F (g \circ \text{Inl}) A * F (g \circ \text{Inr}) B \)
proof –
  have \( A <+> B = \text{Inl } A \uplus \text{Inr } B \) by auto
moreover from fin have \( \text{finite } (\text{Inl } A) \) \( \text{finite } (\text{Inr } B) \) by auto
moreover have \( \text{Inl } A \cap \text{Inr } B = \{\} \) by auto
moreover have \( \text{inj-on } \text{Inl } A \text{ inj-on } \text{Inr } B \) by (auto intro: inj-onI)
ultimately show \(?thesis\)
  using fin by (simp add: union-disjoint reindex)
lemma same-carrier:
  assumes finite C
  assumes subset: A ⊆ C B ⊆ C
  assumes trivial: ∃a. a ∈ C − A → g a = 1 ∃b. b ∈ C − B → h b = 1
  shows F g A = F h B ⟷ F g C = F h C
proof
  have finite A and finite B and finite (C − A) and finite (C − B)
    using ⟨finite C⟩ subset by (auto elim: finite-subset)
  from subset have [simp]: A − (C − A) = A by auto
  from subset have [simp]: B − (C − B) = B by auto
  from subset have C = A ∪ (C − A) by auto
  then have F g C = F g (A ∪ (C − A)) by simp
  also have ... = F g (A − (C − A)) ∗ F g (C − A − A) ∗ F g (A ∩ (C − A))
    using ⟨finite A, finite (C − A)⟩ by (simp only: union-diff2)
  finally have ∗: F g C = F g A using trivial by simp
  from subset have C = B ∪ (C − B) by auto
  then have F h C = F h (B ∪ (C − B)) by simp
  also have ... = F h (B − (C − B)) ∗ F h (C − B − B) ∗ F h (B ∩ (C − B))
    using ⟨finite B, finite (C − B)⟩ by (simp only: union-diff2)
  finally have F h C = F h B
    using trivial by simp
  with ∗ show ?thesis by simp
qed

lemma same-carrier1:
  assumes finite C
  assumes subset: A ⊆ C B ⊆ C
  assumes trivial: ∃a. a ∈ C − A → g a = 1 ∃b. b ∈ C − B → h b = 1
  assumes F g C = F h C
  shows F g A = F h B
  using assms same-carrier [of C A B] by simp
end

2.2 Generalized summation over a set

class ab-semigroup-add-0 = zero + ab-semigroup-add +
  assumes zero-neutral [simp]: x + y + 0 = x + y
  assumes zero-idem [simp]: 0 + 0 = 0
begin

sublocale sum-0: abel-semigroup-set plus 0
  defines sum-0 = sum-0.F
  by unfold-locales simp-all

abbreviation Sum-0 (\sum - [1000] 999)
  where \sum A ≡ sum-0 (λx. x) A

end
context comm-monoid-add
begin
subclass ab-semigroup-add-0 
  by unfold-locales simp-all
end

Now: lots of fancy syntax. First, \( \sum_{x \in A} (\lambda x. e) \) is written \( \sum x \in A. e \).

syntax (ASCII)
\(-\sum :: \text{pttrn} \Rightarrow 'a \set \Rightarrow 'b :: \text{comm-monoid-add} \ ((\exists \text{SUM } (-/)\cdot \cdot ) [0, 51, 10]) \)
syntax
\(-\sum :: \text{pttrn} \Rightarrow 'a \set \Rightarrow 'b :: \text{comm-monoid-add} \ ((\exists \cdot \in\cdot ) [0, 51, 10]) \)
translations — Beware of argument permutation!
\[ \sum_i \in A. b = \text{CONST sum-0 } (\lambda i. b) A \]

Instead of \( \sum x \in \{x. P\}. e \) we introduce the shorter \( \sum x | P. e \).

syntax (ASCII)
\(-\text{qsum} :: \text{pttrn} \Rightarrow \text{bool} \Rightarrow 'a \set \Rightarrow 'a \ ((\exists \text{SUM } -|\cdot | -) [0, 0, 10]) \)
syntax
\(-\text{qsum} :: \text{pttrn} \Rightarrow \text{bool} \Rightarrow 'a \set \Rightarrow 'a \ ((\exists \cdot | (-/)\cdot ) [0, 0, 10]) \)
translations
\[ \sum x | P. t = \Rightarrow \text{CONST sum-0 } (\lambda x. t) \{x. P\} \]

print-translation :
let
fun sum-tr' [Abs (x, Tx, t), Const (@{const-syntax Collect}, -)] $ Abs (y, Ty, P)] =
  if x <> y then raise Match
  else let
  val x' = Syntax-Trans.mark-bound-body (x, Tx);
  val t' = subst-bound (x', t);
  val P' = subst-bound (x', P);
  in Syntax.const @{syntax-const -qsum} $ Syntax-Trans.mark-bound-abs (x, Tx) $ P' $ t' 
  end
in [(@{const-syntax sum-0}, K sum-tr')] end

lemma (in ab-semigroup-add-0) sum-image-gen-0:
  assumes fin: finite S
shows \( \text{sum-0} g \ S = \text{sum-0} (\lambda y. \text{sum-0} g \ \{x. \ x \in S \land f x = y\}) \ (f ' S) \)

proof –

have \( \{y. \ y \in f'S \land f x = y\} = \{f x\} \) if \( x \in S \) for \( x \)

using that by auto

then have \( \text{sum-0} g \ S = \text{sum-0} (\lambda x. \text{sum-0} (\lambda y. g x) \ \{y. \ y \in f'S \land f x = y\}) \ S \)

by simp

also have \( \ldots = \text{sum-0} (\lambda y. \text{sum-0} g \ \{x. \ x \in S \land f x = y\}) \ (f ' S) \)

by (rule \text{sum-0}.\text{swap-restrict} \ [\text{OF fin finite-imageI} \ [\text{OF fin}]])

finally show \( \text{thesis} \).

qed

2.2.1 Properties in more restricted classes of structures

lemma \text{sum-Un2}:

assumes finite \((A \cup B)\)

shows \( \text{sum-0} f \ (A \cup B) = \text{sum-0} f \ (A - B) + \text{sum-0} f \ (B - A) + \text{sum-0} f \ (A \cap B) \)

proof –

have \( A \cup B = A - B \cup (B - A) \cup A \cap B \)

by auto

with assms show \( \text{thesis} \)

by simp (subst \text{sum-0}.\text{union-disjoint}, auto)+

qed

class \text{ordered-ab-semigroup-add-0} = \text{ab-semigroup-add-0} + \text{ordered-ab-semigroup-add}

begin

lemma \text{add-nonneg-nonneg} [simp]: \( \theta \leq a \implies \theta \leq b \implies \theta \leq a + b \)

using \text{add-mono} [of \theta \ a \ b] by simp

lemma \text{add-nonpos-nonpos}: \( a \leq 0 \implies b \leq 0 \implies a + b \leq 0 \)

using \text{add-mono} [of \theta \ a \ b] by simp

end

lemma (in \text{ordered-ab-semigroup-add-0}) \text{sum-mono}:

\( (\bigwedge i. \ i \in K \implies f i \leq g i) \implies (\sum i \in K. f i) \leq (\sum i \in K. g i) \)

by (induct \text{K rule: infinite-finite-induct}) (use \text{add-mono} in auto)

lemma (in \text{ordered-ab-semigroup-add-0}) \text{sum-mono-00}:

\( (\bigwedge i. \ i \in K \implies f i + \theta \leq g i + \theta) \implies (\sum i \in K. f i) \leq (\sum i \in K. g i) \)

proof (induct \text{K rule: infinite-finite-induct})

\text{case} (\text{infinite} \ A)

then show \( \text{thesis} \) by simp

next

\text{case} \text{empty}

then show \( \text{thesis} \) by simp

next
case (insert x F)
then show ?case
proof -
  fix x :: 'b and F :: 'b set
  assume a1: finite F
  assume a2: x \notin F
  assume a3: (\forall i. i \in F \Rightarrow f i + 0 \leq g i + 0) \Rightarrow \text{sum-0 } f F \leq \text{sum-0 } g F
  assume a4: \forall i. i \in insert x F \Rightarrow f i + 0 \leq g i + 0
  obtain bb :: 'b where
    f5: bb \in F \land \neg f bb + 0 \leq g bb + 0 \lor \text{sum-0 } f F \leq \text{sum-0 } g F
    using a3 by blast
  have \forall b. x \neq b \lor f b + 0 \leq g b + 0
    using a4 by simp
  then have \forall a aa. f x + 0 + a \leq g x + 0 + aa \lor a \leq aa
    using add-mono by blast
  then show \text{sum-0 } f (insert x F) \leq \text{sum-0 } g (insert x F)
    using f5 a4 a2 a1 by (metis (no-types) add-assoc insert-iff sum-0.insert sum-0.one-F)
qed

lemma (in ordered-ab-semigroup-add-0) sum-mono-0:
(\forall i. i \in K \Rightarrow f i + 0 \leq g i) \Rightarrow (\sum i \in K. f i) \leq (\sum i \in K. g i)
apply (rule sum-mono-00)
by (metis add-right-mono zero-neutral)

context ordered-ab-semigroup-add-0
begin

lemma sum-nonneg: (\forall x. x \in A \Rightarrow 0 \leq f x) \Rightarrow 0 \leq \text{sum-0 } f A
proof (induct A rule: infinite-finite-induct)
case infinite
  then show ?case by simp
next
case empty
  then show ?case by simp
next
case (insert x F)
  then have 0 + 0 \leq f x + \text{sum-0 } f F by (blast intro: add-mono)
  with insert show ?case by simp
qed

lemma sum-nonpos: (\forall x. x \in A \Rightarrow f x \leq 0) \Rightarrow \text{sum-0 } f A \leq 0
proof (induct A rule: infinite-finite-induct)
case infinite
  then show ?case by simp
next
case empty
  then show ?case by simp
next
  case (insert x F)
  then have \( f x + \sum 0 f F \leq 0 + 0 \) by (blast intro: add-mono)
  with insert show \( \text{case by simp} \)
  qed

lemma sum-mono2:
  assumes fin: finite B
  and sub: \( A \subseteq B \)
  and nn: \( \forall b. b \in B - A \implies 0 \leq f b \)
  shows \( \sum 0 f A \leq \sum 0 f B \)
  proof
    have \( \sum 0 f A \leq \sum 0 f A + \sum 0 f (B - A) \)
      by (metis add-left-mono sum-0 F-one nn sum-nonneg)
    also from fin finite-subset[OF sub fin]
      have \( \ldots = \sum 0 f (A \cup (B - A)) \)
      by (simp add: sum-0.union-disjoint del: Un-Diff-cancel)
    also from sub have \( A \cup (B - A) = B \) by blast
    finally show \( \text{thesis} \).
    qed

lemma sum-le-included:
  assumes finite s finite t
  and \( \forall y \in t. 0 \leq g y \) \( \forall x \in s. \exists y \in t. i y = x \land f x \leq g y \)
  shows \( \sum 0 f s \leq \sum 0 g t \)
  proof
    have \( \sum 0 f s \leq \sum 0 (\lambda y. \sum 0 g \{ x. x \in t \land i x = y \}) s \)
      (rule sum-mono-0)
      fix y
      assume y \( \in s \)
      with assms obtain z where \( z \in t y = i z f y \leq g z \) by auto
      hence \( f y + 0 \leq \sum 0 g \{ z \} \)
        by (metis Diff-eq-empty-iff add-commute finite.simps add-left-mono
          sum-0.empty sum-0.insert-remove subset-insert1)
      also have \( \ldots \leq \sum 0 g \{ x \in t. i x = y \} \)
        apply (rule sum-mono2)
        using assms z by simp-all
      finally show \( f y + 0 \leq \sum 0 g \{ x \in t. i x = y \} \).
    qed
  also have \( \ldots \leq \sum 0 (\lambda y. \sum 0 g \{ x. x \in t \land i x = y \}) (i \cdot t) \)
    using assms(2-4) by (auto intro!: sum-mono2 sum-nonneg)
  also have \( \ldots \leq \sum 0 g t \)
    using assms by (auto simp: sum-image-gen-0[symmetric])
  finally show \( \text{thesis} \).
  qed

end

lemma sum-comp-morphism:
  \( h 0 = 0 \implies (\forall x y. h (x + y) = h x + h y) \implies \sum 0 (h \circ g) A = h (\sum 0 A) \)
lemma sum-cong-Suc:
  assumes \( \emptyset \notin A \land \forall x. \text{Suc} \ x \in A \implies f (\text{Suc} \ x) = g (\text{Suc} \ x) \)
  shows \( \text{sum-0} \ f A = \text{sum-0} \ g A \)
proof (rule sum-0.cong)
  fix \( x \)
  assume \( x \in A \)
  with assms(1) show \( f \ x = g \ x \)
    by (cases \( x \)) (auto intro!: assms(2))
qed simp-all
end

3 Algebras for Aggregation and Minimisation

This theory gives algebras with operations for aggregation and minimisation. In the weighted-graph model of matrices over (extended) numbers, the operations have the following meaning. The binary operation + adds the weights of corresponding edges of two graphs. Addition does not have to be the standard addition on numbers, but can be any aggregation satisfying certain basic properties as demonstrated by various models of the algebras in another theory. The unary operation sum adds the weights of all edges of a graph. The result is a single aggregated weight using the same aggregation as + but applied internally to the edges of a single graph. The unary operation minarc finds an edge with a minimal weight in a graph. It yields the position of such an edge as a regular element of a Stone relation algebra.

We give axioms for these operations which are sufficient to prove the correctness of Prim’s and Kruskal’s minimum spanning tree algorithms. The operations have been proposed and axiomatised first in [1] with simplified axioms given in [4]. The present version adds two axioms to prove total correctness of the spanning tree algorithms as discussed in [5].

theory Aggregation-Algebras


begin

context sup

begin

no-notation
  sup (infixl + 65)
We first introduce s-algebras as a class with the operations $+$ and $\text{sum}$. Axiom $\text{sum-plus-right-isotone}$ states that for non-empty graphs, the operation $+$ is $\leq$-isotone in its second argument on the image of the aggregation operation $\text{sum}$. Axiom $\text{sum-bot}$ expresses that the empty graph contributes no weight. Axiom $\text{sum-plus}$ generalises the inclusion-exclusion principle to sets of weights. Axiom $\text{sum-conv}$ specifies that reversing edge directions does not change the aggregated weight. In instances of $s$-algebra, aggregated weights can be partially ordered.

```plaintext
class sum =
  fixes sum :: 'a ⇒ 'a

class s-algebra =
  stone-relation-algebra +
  plus + sum +
  assumes sum-plus-right-isotone: $x \neq \text{bot}$ ∧ $\text{sum } x \leq \text{sum } y$ → $\text{sum } x + \text{sum } y$
  ≤ $\text{sum } z + \text{sum } y$
  assumes sum-bot: $\text{sum } x + \text{sum } \text{bot} = \text{sum } x$
  assumes sum-plus: $\text{sum } x + \text{sum } y = \text{sum } (x \sqcup y) + \text{sum } (x \sqcap y)$
  assumes sum-conv: $\text{sum } (x^T) = \text{sum } x$

begin

lemma sum-disjoint:
  assumes $x \sqcap y = \text{bot}$
  shows $\text{sum } ((x \sqcup y) \sqcap z) = \text{sum } (x \sqcap z) + \text{sum } (y \sqcap z)$
  by (subst sum-plus) (metis assms inf-sup-monoid.add-assoc inf-sup-monoid.add-commute inf-bot-left inf-sup-distrib2 sum-bot)

lemma sum-disjoint-3:
  assumes $w \sqcap x = \text{bot}$
  and $w \sqcap y = \text{bot}$
  and $x \sqcap y = \text{bot}$
  shows $\text{sum } ((w \sqcup x \sqcup y) \sqcap z) = \text{sum } (w \sqcap z) + \text{sum } (x \sqcap z) + \text{sum } (y \sqcap z)$
  by (metis assms inf-sup-distrib2 sup-idem sum-disjoint)

lemma sum-symmetric:
  assumes $y = y^T$
  shows $\text{sum } (x^T \sqcap y) = \text{sum } (x \sqcap y)$
  by (metis assms sum-cone cone-dist-inf)

lemma sum-commute:
  $\text{sum } x + \text{sum } y = \text{sum } y + \text{sum } x$
```
by (metis inf-commute sum-plus sup-commute)

end

We next introduce the operation \textit{minarc}. Axiom \textit{minarc-below} expresses that the result of \textit{minarc} is contained in the graph ignoring the weights. Axiom \textit{minarc-arc} states that the result of \textit{minarc} is a single unweighted edge if the graph is not empty. Axiom \textit{minarc-min} specifies that any edge in the graph weighs at least as much as the edge at the position indicated by the result of \textit{minarc}, where weights of edges between different nodes are compared by applying the operation \textit{sum} to single-edge graphs. Axiom \textit{sum-linear} requires that aggregated weights are linearly ordered, which is necessary for both Prim’s and Kruskal’s minimum spanning tree algorithms. Axiom \textit{finite-regular} ensures that there are only finitely many unweighted graphs, and therefore only finitely many edges and nodes in a graph; again this is necessary for the minimum spanning tree algorithms we consider.

class \textit{minarc} =
  fixes \textit{minarc} :: 'a ⇒ 'a

class \textit{m-algebra} = \textit{s-algebra} + \textit{minarc} +
  assumes \textit{minarc-below}: \textit{minarc} x ≤ -- x
  assumes \textit{minarc-arc}: x ≠ bot → arc (\textit{minarc} x)
  assumes \textit{minarc-min}: arc y ∧ y ∩ x ≠ bot → sum (\textit{minarc} x ∩ x) ≤ sum (y ∩ x)
  assumes \textit{sum-linear}: sum x ≤ sum y ∨ sum y ≤ sum x
  assumes \textit{finite-regular}: finite \{ x . regular x \}

begin

Axioms \textit{minarc-below} and \textit{minarc-arc} suffice to derive the Tarski rule in Stone relation algebras.

subclass \textit{stone-relation-algebra-tarski}

proof unfold-locales
  fix x
  let ?a = \textit{minarc} x
  assume 1: regular x
  assume x ≠ bot
  hence arc ?a
    by (simp add: \textit{minarc-arc})
  hence top = top ∗ ?a ∗ top
    by (simp add: comp-associative)
  also have ... ≤ top ∗ -- x ∗ top
    by (simp add: \textit{minarc-below} mult-isotone)
  finally show top ∗ x ∗ top = top
    using 1 antisym by simp
qed

lemma \textit{minarc-bot}:
  \textit{minarc} bot = bot
by (metis bot-unique minarc-below regular-closed-bot)

lemma minarc-bot-iff:
  minarc \( x = \text{bot} \) \( \iff \) \( x = \text{bot} \)
using covector-bot-closed inf-bot-right minarc-arc vector-bot-closed minarc-bot
by fastforce

lemma minarc-meet-bot:
  assumes minarc \( x \cap x = \text{bot} \)
  shows minarc \( x = \text{bot} \)
proof
  have minarc \( x \leq -x \)
  using assms pseudo-complement by auto
  thus \(?thesis\)
  by (metis minarc-below inf-absorb1 inf-import-p inf-p)
qed

lemma minarc-meet-bot-minarc-iff:
  minarc \( x \cap x = \text{bot} \) \( \iff \) minarc \( x = \text{bot} \)
using comp-inf semiring mult-not-zero minarc-meet-bot
by blast

lemma minarc-meet-bot-iff:
  minarc x \( \cap \) x = bot \( \iff \) minarc x = bot
using inf-bot-right minarc-bot-iff minarc-meet-bot by blast

lemma minarc-regular:
  regular (minarc x)
proof (cases x = bot)
  assume x = bot
  thus \(?thesis\)
  by (simp add: minarc-bot)
next
  assume x \( \neq \) bot
  thus \(?thesis\)
  by (simp add: arc-regular minarc-arc)
qed

lemma minarc-selection:
  selection (minarc x \( \cap \) y) y
using inf-assoc minarc-regular selection-closed-id by auto

lemma minarc-below-regular:
  regular x \( \implies \) minarc x \( \leq \) x
by (metis minarc-below)

end
4 Matrix Algebras for Aggregation and Minimisation

This theory formalises aggregation orders and lattices as introduced in [4]. Aggregation in these algebras is an associative and commutative operation satisfying additional properties related to the partial order and its least element. We apply the aggregation operation to finite matrices over the aggregation algebras, which shows that they form an s-algebra. By requiring the aggregation algebras to be linearly ordered, we also obtain that the matrices form an m-algebra.

This is an intermediate step in demonstrating that weighted graphs form an s-algebra and an m-algebra. The present theory specifies abstract properties for the edge weights and shows that matrices over such weights form an instance of s-algebras and m-algebras. A second step taken in a separate theory gives concrete instances of edge weights satisfying the abstract properties introduced here.

Lifting the aggregation to matrices requires finite sums over elements taken from commutative semigroups with an element that is a unit on the image of the semigroup operation. Because standard sums assume a commutative monoid we have to derive a number of properties of these generalised sums as their statements or proofs are different.

theory Matrix-Aggregation-Algebras


begin

no-notation
inf (infixl \(\sqcap\))
and uminus (\(-\ [81] 80\))

4.1 Aggregation Orders and Finite Sums

An aggregation order is a partial order with a least element and an associative commutative operation satisfying certain properties. Axiom add-add-bot introduces almost a commutative monoid; we require that bot is a unit only on the image of the aggregation operation. This is necessary since it is not a unit of a number of aggregation operations we are interested in. Axiom add-right-isotone states that aggregation is \(\leq\)-isotone on the image of the aggregation operation. Its assumption \(x \not\approx \) bot is necessary because the in-
troduction of new edges can decrease the aggregated value. Axiom \textit{add-bot} expresses that aggregation is zero-sum-free.

\begin{verbatim}
class aggregation-order = order-bot + ab-semigroup-add +  
  assumes add-right-isotone: \( x \neq \text{bot} \land x + \text{bot} \leq y + \text{bot} \rightarrow x + z \leq y + z \)  
  assumes add-add-bot [simp]: \( x + y + \text{bot} = x + y \)  
  assumes add-bot: \( x + y = \text{bot} \rightarrow x = \text{bot} \)

begin

abbreviation zero \( \equiv \text{bot} + \text{bot} \)

sublocale aggregation: ab-semigroup-add-0 where plus = plus and zero = zero
  apply unfold-locales
  using add-assoc add-add-bot by auto

lemma add-bot-bot-bot: 
  \( x + \text{bot} + \text{bot} + \text{bot} = x + \text{bot} \)
  by simp

end

We introduce notation for finite sums over aggregation orders. The index variable of the summation ranges over the finite universe of its type. Finite sums are defined recursively using the binary aggregation and \( \bot + \bot \) for the empty sum.

\begin{verbatim}
syntax (xsymbols)
  \( \sum_{a} \) \( \cdot \) \( \cdot \) :: idt \Rightarrow 'a::bounded-semilattice-sup-bot \Rightarrow 'a ((\( \sum_{a} \) \( \cdot \) \( \cdot \)) [0,10] 10)

translations
  \( \sum_{x} t \Rightarrow XCONST \text{ab-semigroup-add-0}.\text{sum}-0 XCONST \text{plus} (XCONST \text{plus} XCONST \text{bot} XCONST \text{bot}) (\lambda x . t) \{ x . \text{CONST True } \}

The following are basic properties of such sums.

lemma agg-sum-bot: 
  \( (\sum_{k} \text{bot}::'a::aggregation-order) = \text{bot} + \text{bot} \)
proof \( \text{(induct rule: infinite-finite-induct)} \)
  case \( \text{infinite A} \)
  thus \( ?\text{case} \)
  by simp
  next
  case \text{empty}
  thus \( ?\text{case} \)
  by simp
  next
  case \( \text{insert x F} \)
  thus \( ?\text{case} \)
  by \( \text{(metis add.commute add-add-bot aggregation.sum-0.insert)} \)
qed
\end{verbatim}

23
lemma agg-sum-bot-bot:
\[(\sum_k \text{bot} + \text{bot} : ':a::{\text{aggregation-order}}) = \text{bot} + \text{bot}\]
by (rule aggregation.sum-0.neutral-const)

lemma agg-sum-not-bot-1:
fixes f :: ':a::{\text{finite}} \Rightarrow ':b::{\text{aggregation-order}}
assumes f i \neq \text{bot}
shows \((\sum_k f k) \neq \text{bot}\)
by (metis assms add-bot aggregation.sum-0.remove finite-code mem-Collect-eq)

lemma agg-sum-not-bot:
fixes f :: '(':a::{\text{finite}}, ':b::{\text{aggregation-order}}) square
assumes f (i,j) \neq \text{bot}
shows \((\sum_k \sum_l f (k,l)) \neq \text{bot}\)
by (rule aggregation.sum-0.distrib)

lemma agg-sum-distrib:
fixes f :: ':a \Rightarrow ':b::{\text{aggregation-order}}
square
assumes f k + g k = (\sum_k f k) + (\sum_k g k)
shows \((\sum_k f k + g k) = (\sum_k f k) + (\sum_k g k)\)
by (rule aggregation.sum-0.distrib)

lemma agg-sum-distrib-2:
fixes f :: '(':a,:b::{\text{aggregation-order}}) square
assumes f k + g k = (\sum_k f k) + (\sum_k g k)
shows \((\sum_k \sum_l f (k,l) + g (k,l)) = (\sum_k \sum_l f (k,l)) + (\sum_k \sum_l g (k,l))\)
proof
have \((\sum_k \sum_l f (k,l) + g (k,l)) = (\sum_k \sum_l f (k,l) + (\sum_l g (k,l)))\)
by (metis (no-types) aggregation.sum-0.distrib)
also have \(\ldots = (\sum_k \sum_l f (k,l)) + (\sum_k \sum_l g (k,l))\)
by (metis (no-types) aggregation.sum-0.distrib)
finally show \(\text{thesis}\)
qed

lemma agg-sum-add-bot:
fixes f :: ':a \Rightarrow ':b::{\text{aggregation-order}}
square
assumes \((\sum_k f k + \text{bot}) = (\sum_k f k) + \text{bot}\)
shows \((\sum_k f k + \text{bot}) = (\sum_k f k) + \text{bot}\)
by (metis (no-types) add-add-bot aggregation.sum-0.F-one)

lemma agg-sum-add-bot-2:
fixes f :: ':a \Rightarrow ':b::{\text{aggregation-order}}
square
assumes \((\sum_k f k + \text{bot}) = (\sum_k f k) + \text{bot}\)
shows \((\sum_k f k + \text{bot}) = (\sum_k f k) + \text{bot}\)
proof
have \((\sum_k f k + \text{bot}) = (\sum_k f k) + (\sum_k :':a::bot :':b)\)
using agg-sum-distrib by simp
also have \(\ldots = (\sum_k f k) + (\text{bot} + \text{bot})\)
by (metis agg-sum-bot)
also have \(\ldots = (\sum_k f k)\)
by simp

24
finally show thesis
  by simp
qed

lemma agg-sum-commute:
  fixes f :: ('a,'b::aggregation-order) square
  shows \( \sum_k \sum_l f(k,l) = \sum_l \sum_k f(k,l) \)
  by (rule aggregation.sum-0.swap)

lemma agg-delta:
  fixes f :: 'a::finite => 'b::aggregation-order
  shows \( \sum_l (\text{if } l = j \text{ then } f(l) \text{ else } 0) \) = f(j) + bot
  apply (subst aggregation.sum-0.delta)
  apply simp
  by (metis add.commute add-left-commute add-add-bot mem-Collect-eq)

lemma agg-delta-1:
  fixes f :: 'a::finite => 'b::aggregation-order
  shows \( \sum_l (\text{if } l = j \text{ then } f(l) \text{ else } bot) \) = f(j) + bot
  proof -
    let ?f = (\lambda l. \text{if } l = j \text{ then } f(l) \text{ else } bot)
    let ?S = \{ l::'a . True \}
    show thesis
      proof (cases j \in ?S)
        case False
        thus thesis by simp
      next
        case True
        let ?A = ?S - \{ j \}
        let ?B = \{ j \}
        from True have eq: ?S = ?A \cup ?B
          by blast
        have dj: ?A \cap ?B = \{ \}
          by simp
        have fAB: finite ?A finite ?B
          by auto
        have aggregation.sum-0 ?f ?S = aggregation.sum-0 ?f ?A + aggregation.sum-0 ?f ?B
          using aggregation.sum-0.union-disjoint[of fAB dj, of ?f, unfolded eq [symmetric]] by simp
        also have ... = aggregation.sum-0 (\lambda l . bot) ?A + aggregation.sum-0 ?f ?B
          by (subst aggregation.sum-0.cong[where ?B=?A]) simp-all
        also have ... = zero + aggregation.sum-0 ?f ?B
          by (metis (no-types, lifting) add.commute add-add-bot aggregation.sum-0.F-g-one aggregation.sum-0.neutral)
        also have ... = \text{zero + (f\( j \) + zero)}
          by simp
        also have ... = \text{f\( j \) + bot}
          by (metis add.commute add-left-commute add-add-bot)
finally show \( \text{thesis} \).

qed

qed

**lemma** agg-delta-2:

fixes \( f :: (\text{'}a::\text{finite},\text{'}b::\text{aggregation-order}) \text{ square} \)

shows \( (\sum_k \sum_l \text{ if } k = i \land l = j \text{ then } f (k,l) \text{ else } \text{bot}) = f (i,j) + \text{bot} \)

proof –

have \( \forall k . (\sum_l \text{ if } k = i \land l = j \text{ then } f (k,l) \text{ else } \text{bot}) = (\sum_l \text{ if } l = j \text{ then } \text{if } k = i \text{ then } f (k,l) + \text{bot} \text{ else } \text{zero}) \)

by meson

also have \( \ldots = (\text{if } k = i \text{ then } f (k,j) \text{ else } \text{bot}) + \text{bot} \)

by (rule agg-delta-1)

finally show \( (\sum_l \text{ if } k = i \land l = j \text{ then } f (k,l) \text{ else } \text{bot}) = (\text{if } k = i \text{ then } f (k,j) + \text{bot} \text{ else } \text{zero}) \)

by simp

qed

hence \( (\sum_k \sum_l \text{ if } k = i \land l = j \text{ then } f (k,l) \text{ else } \text{bot}) = (\sum_k \text{ if } k = i \text{ then } f (k,j) + \text{bot} \text{ else } \text{zero}) \)

using aggregation.sum-0.cong by auto

also have \( \ldots = f (i,j) + \text{bot} \)

apply (subst agg-delta)

by simp

finally show \( \text{thesis} \).

qed

4.2 Matrix Aggregation

The following definitions introduce the matrix of unit elements, component-wise aggregation and aggregation on matrices. The aggregation of a matrix is a single value, but because s-algebras are single-sorted the result has to be encoded as a matrix of the same type (size) as the input. We store the aggregated matrix value in the ‘first’ entry of a matrix, setting all other entries to the unit value. The first entry is determined by requiring an enumeration of indices. It does not have to be the first entry; any fixed location in the matrix would work as well.

**definition** zero-matrix :: (\( \text{'}a,\text{'}b::\{\text{plus,bot}\} \)) \text{ square} \ (mzero) \ where \ mzero = (\lambda e . \text{bot} + \text{bot})

**definition** plus-matrix :: (\( \text{'}a,\text{'}b::\text{plus} \)) \Rightarrow (\( \text{'}a,\text{'}b \)) \text{ square} \Rightarrow (\( \text{'}a,\text{'}b \)) \text{ square} \ (\text{infixl} \oplus_{M} 65) \ where \ plus-matrix \ f \ g = (\lambda e . \ f \ e + \ g \ e)
definition sum-matrix :: ('a::enum,'b::{plus,bot}) square ⇒ ('a,'b) square

(sum_M - [80] 80) where sum-matrix f = (λ(i,j). if i = hd enum-class.enum ∧ j = i then ∑ k ∑ l f (k,l) else bot + bot)

Basic properties of these operations are given in the following.

lemma bot-plus-bot:

mbot ⊕ M mbot = mzero
by (simp add: plus-matrix-def bot-matrix-def zero-matrix-def)

lemma sum-bot:

sum_M (mbot :: ('a::enum,'b::aggregation-order) square) = mzero
proof (rule ext, rule prod-cases)
fix i j :: 'a
have (sum_M mbot :: ('a,'b) square) (i,j) = (if i = hd enum-class.enum ∧ j = i then ∑ (k::'a) ∑ (l::'a) bot else bot + bot)
  by (unfold sum-matrix-def bot-matrix-def) simp
also have ... = bot + bot
  using agg-sum-bot aggregation.sum-0.neutral by fastforce
also have ...
  by (simp add: zero-matrix-def)
finally show (sum_M mbot :: ('a,'b) square) (i,j) = mzero (i,j)
qed

lemma sum-plus-bot:

fixes f :: ('a::enum,'b::aggregation-order) square
shows sum_M f ⊕_M mbot = sum_M f
proof (rule ext, rule prod-cases)
let ?h = hd enum-class.enum
fix i j
have (sum_M f ⊕_M mbot) (i,j) = (if i = ?h ∧ j = i then (∑ k ∑ l f (k,l)) + bot else zero + bot)
  by (simp add: plus-matrix-def bot-matrix-def sum-matrix-def)
also have ...
  by (metis (no-types, lifting) add-add-bot aggregation.sum-0.F-one)
also have ...
  by (simp add: sum-matrix-def)
finally show (sum_M f ⊕_M mbot) (i,j) = (sum_M f) (i,j)
  by simp
qed

lemma sum-plus-zero:

fixes f :: ('a::enum,'b::aggregation-order) square
shows sum_M f ⊕_M mzero = sum_M f
by (rule ext, rule prod-cases) (simp add: plus-matrix-def zero-matrix-def sum-matrix-def)

lemma agg-matrix-bot:

fixes f :: ('a,'b::aggregation-order) square
assumes $\forall i \ j. \ f(i,j) = \text{bot}$
shows $f = \text{mbot}$
apply (unfold bot-matrix-def)
using assms by auto

We consider a different implementation of matrix aggregation which stores the aggregated value in all entries of the matrix instead of a particular one. This does not require an enumeration of the indices. All results continue to hold using this alternative implementation.

definition sum-matrix-2 :: $(\alpha, \beta :: \{\text{plus}, \text{bot}\}) \square \Rightarrow (\alpha, \beta) \square$ where sum-matrix-2 $f = (\lambda e. \sum_k \sum_l f(k,l))$

lemma sum-bot-2:
sum$_2$ $M$ $(\text{mbot} :: (\alpha, \beta :: \text{aggregation-order}) \square)$ = $\text{mzero}$
proof
fix $e$
have $(\sum_k \sum_l f(k,l)) + \text{bot}$
  by (simp add: plus-matrix-def bot-matrix-def sum-matrix-2-def)
also have $\ldots = \text{bot} + \text{bot}$
  using agg-sum-bot aggregation.sum-0.neutral by fastforce
also have $\ldots = \text{mzero} e$
  by simp
finally show $(\sum_k \sum_l f(k,l))$ = $\text{mzero} e$
qed

lemma sum-plus-bot-2:
fixes $f :: (\alpha, \beta :: \text{aggregation-order}) \square$
shows sum$_2$ $M$ $f \oplus M \text{mbot} = $ sum$_2$ $M$ $f$
proof
fix $e$
have $(\sum_k \sum_l f(k,l)) + \text{bot}$
  by (simp add: plus-matrix-def bot-matrix-def sum-matrix-2-def)
also have $\ldots = (\sum_k \sum_l f(k,l))$
  by (metis no-types, lifting add-add-bot aggregation.sum-0.F-one)
also have $\ldots = (\text{sum}_2 M f)$
  by (simp add: sum-matrix-2-def)
finally show $(\sum_k \sum_l f(k,l))$ = $(\text{sum}_2 M f)$
  by simp
qed

lemma sum-plus-zero-2:
fixes $f :: (\alpha, \beta :: \text{aggregation-order}) \square$
shows sum$_2$ $M$ $f \oplus M \text{mzero} = $ sum$_2$ $M$ $f$
by (simp add: plus-matrix-def zero-matrix-def sum-matrix-2-def)

4.3 Aggregation Lattices
We extend aggregation orders to dense bounded distributive lattices. Axiom 
\textit{add-lattice} implements the inclusion-exclusion principle at the level of edge 
weights.

class \textit{aggregation-lattice} = \textit{bounded-distrib-lattice} + \textit{dense-lattice} + 
\textit{aggregation-order} + 
assumes \textit{add-lattice}: \( x + y = (x \sqcup y) + (x \sqcap y) \)

Aggregation lattices form a Stone relation algebra by reusing the meet 
operation as composition, using identity as converse and a standard imple-
mentation of pseudocomplement.

class \textit{aggregation-algebra} = \textit{aggregation-lattice} + \textit{uminus} + \textit{one} + \textit{times} + \textit{conv} + 
assumes \textit{uminus-def} [simp]: \(-x = (if \text{ } x = \text{bot} \text{ then top else bot})\) 
assumes \textit{one-def} [simp]: \(1 = \text{top}\) 
assumes \textit{times-def} [simp]: \(x \ast y = x \sqcap y\) 
assumes \textit{conv-def} [simp]: \(x^T = x\)

begin 
subclass \textit{stone-algebra} 
apply unfold-locales 
using bot-meet-irreducible bot-unique by auto 
subclass \textit{stone-relation-algebra} 
apply unfold-locales 
prefer 9 using bot-meet-irreducible 
apply auto[1] 
by (simp-all add: inf.assoc le-infI2 inf-sup-distrib1 inf-sup-distrib2 inf.commute inf.left-commute)
end

We show that matrices over aggregation lattices form an s-algebra using 
the above operations.

interpretation \textit{agg-square-s-algebra}: s-algebra where sup = sup-matrix and inf = inf-matrix and less-eq = less-eq-matrix and less = less-matrix and bot = bot-matrix:(\'a::enum,\'b::aggregation-algebra) square and top = top-matrix and 
uminus = uminus-matrix and one = one-matrix and times = times-matrix and 
conv = conv-matrix and plus = plus-matrix and sum = sum-matrix

proof 
fix \( f \) \( g \) \( h \) :: (\'a,\'b) square 
show \( f \neq \text{mbot} \land \sum_M f \preceq \sum_M g \longrightarrow h \oplus_M \sum_M f \preceq h \oplus_M \sum_M g \)

proof 
let \(?h = \text{hd enum-class.enum} \)
assume 1: \( f \neq \text{mbot} \land \sum_M f \preceq \sum_M g \)
hence \( \exists k \cdot f(k,l) \neq \text{bot} \)
by (meson agg-matrix-bot)
hence 2: \( \sum_k \sum_l f(k,l) \neq \text{bot} \)
using agg-sum-not-bot by blast
have \( (\sum_k \sum_l f(k,l)) = (\sum_M f) (\text{?h,?h}) \)
by (simp add: sum-matrix-def)
also have ... ≤ (\text{sum}_M g) \cdot (?h, ?h)
using \text{I} by (simp add: less-eq-matrix-def)
also have ... = (\textstyle \sum_k \textstyle \sum_l g (k,l))
by (simp add: sum-matrix-def)
finally have (\textstyle \sum_k \textstyle \sum_l f (k,l)) ≤ (\textstyle \sum_k \textstyle \sum_l g (k,l))
by simp
hence 3: (\textstyle \sum_k \textstyle \sum_l f (k,l)) + \text{bot} ≤ (\textstyle \sum_k \textstyle \sum_l g (k,l)) + \text{bot}
by (metis (no-types, lifting) add-add-bot aggregation.sum-0.F-one)
show \text{h} \oplus_M \text{sum}_M f ≤ \text{h} \oplus_M \text{sum}_M g
proof (unfold less-eq-matrix-def, rule allI, rule prod-cases, unfold plus-matrix-def)
fix \(i, j\)
have 4: \(\text{h} (i,j) + (\textstyle \sum_k \textstyle \sum_l f (k,l)) ≤ \text{h} (i,j) + (\textstyle \sum_k \textstyle \sum_l g (k,l))\)
using 2 3 by (metis (no-types, lifting) add-right-isotone add-commute)
have \(\text{h} (i,j) + (\text{sum}_M f) (i,j) = \text{h} (i,j) + (\text{if} \ i = ?h \land j = i \ 	ext{then} \ \text{sum}_k \text{sum}_l f (k,l) \ 	ext{else} \ 0)\)
by (simp add: sum-matrix-def)
also have ... = (if \ i = ?h \land j = i \ 	ext{then} \ \text{h} (i,j) + (\textstyle \sum_k \textstyle \sum_l f (k,l)) \ 	ext{else} \ h (i,j) + \text{zero})
by simp
also have ... ≤ (if \ i = ?h \land j = i \ 	ext{then} \ \text{h} (i,j) + (\textstyle \sum_k \textstyle \sum_l f (k,l)) \ 	ext{else} \ h (i,j) + \text{zero})
using 4 inf.eq_iff by auto
also have ... = \text{h} (i,j) + (\text{if} \ i = ?h \land j = i \ 	ext{then} \ \text{sum}_k \text{sum}_l g (k,l) \ 	ext{else} \ 0)
by simp
finally show \(\text{h} (i,j) + (\text{sum}_M f) (i,j) ≤ \text{h} (i,j) + (\text{sum}_M g) (i,j)\)
by (simp add: sum-matrix-def)
qed
qed
next
fix \(f :: (\text{a}', \text{b})\) square
show \text{sum}_M f \oplus_M \text{sum}_M \text{mbot} = \text{sum}_M f
by (simp add: sum-bot sum-plus-zero)
next
fix \(f g :: (\text{a}', \text{b})\) square
show \text{sum}_M f \oplus_M \text{sum}_M g = \text{sum}_M (f \oplus g) \oplus_M \text{sum}_M (f \oplus g)
proof (rule ext, rule prod-cases)
fix \(i, j\)
let \(\text{h} = \text{hd enum-class}.	ext{enum}\)
have (\text{sum}_M f \oplus_M \text{sum}_M g) (i,j) = (\text{sum}_M f) (i,j) + (\text{sum}_M g) (i,j)
by (simp add: plus-matrix-def)
also have ... = (\text{if} \ i = ?h \land j = i \ 	ext{then} \ \text{sum}_k \text{sum}_l f (k,l) \ 	ext{else} \ 0) + (\text{if} \ i = ?h \land j = i \ 	ext{then} \ \text{sum}_k \text{sum}_l g (k,l) \ 	ext{else} \ 0)
by (simp add: plus-matrix-def)
also have ... = (\text{if} \ i = ?h \land j = i \ 	ext{then} \ (\text{sum}_k \text{sum}_l f (k,l)) + (\text{sum}_k \text{sum}_l g (k,l)) \ 	ext{else} \ 0)
by simp
also have ... = (\text{if} \ i = ?h \land j = i \ 	ext{then} \ \text{sum}_k \text{sum}_l f (k,l) + g (k,l) \ 	ext{else} \ 0)

30
using `agg-sum-distrib-2` by (metis (no-types))
also have ... = (if i = ?h ∧ j = i then ∑ k ∑ l (f (k,l) ⊕ g (k,l)) + (f (k,l) ∩ g (k,l)) else zero)
using `add-lattice aggregation.sum-0.cong` by (metis (no-types, lifting))
also have ... = (if i = ?h ∧ j = i then ∑ k ∑ l (f ⊕ g) (k,l)) + (f ⊗ g) (k,l) else zero)
by (simp add: `sup-matrix-def inf-matrix-def`)
also have ... = (if i = ?h ∧ j = i then ∑ k ∑ l (f ⊗ g) (k,l)) else zero) + (if i = ?h ∧ j = i then ∑ k ∑ l (f ⊗ g) (k,l) else zero)
by simp
also have ... = (sum_M (f ⊕ g)) (i,j) + (sum_M (f ⊗ g)) (i,j)
by (simp add: `sum-matrix-def`) also have ... = (sum_M (f ⊕ g) ⊔ M sum_M (f ⊗ g)) (i,j)
by (simp add: `plus-matrix-def` finally show (sum_M f ⊔ M sum_M g) (i,j) = (sum_M (f ⊕ g) ⊔ M sum_M (f ⊗ g)) (i,j)
).

qed

next
fix `f :: ('a,b) square`
show `sum_M (f') = sum_M f`
proof (rule ext, rule prod-cases)
fix `i j`
let `?h = hd enum-class.enum`
have (sum_M (f')) (i,j) = (if i = ?h ∧ j = i then ∑ k ∑ l (f') (k,l) else zero) by (simp add: `sum-matrix-def`)
also have ... = (if i = ?h ∧ j = i then ∑ k ∑ l (f (l,k))T else zero) by (simp add: `conv-matrix-def`)
also have ... = (if i = ?h ∧ j = i then ∑ k ∑ l f (l,k) else zero) by simp
also have ... = (if i = ?h ∧ j = i then ∑ l ∑ k f (l,k) else zero) by (metis `agg-sum-commute`)
also have ... = (sum_M f) (i,j)
by (simp add: `sum-matrix-def` finally show (sum_M f') (i,j) = (sum_M f) (i,j)
).

qed

We show the same for the alternative implementation that stores the result of aggregation in all elements of the matrix.

interpretation `agg-square-s-algebra-2`: `s-algebra` where `sup = sup-matrix` and `inf = inf-matrix` and `less-eq = less-eq-matrix` and `less = less-matrix` and `bot = bot-matrix::('a::finite,'b::aggregation-algebra) square` and `top = top-matrix` and `uminus = uminus-matrix` and `one = one-matrix` and `times = times-matrix` and `conv = conv-matrix` and `plus = plus-matrix` and `sum = sum-matrix-2`
proof
  \[ \text{fix } f \, g \, h :: ('a,'b) \text{ square} \]
  \[ \text{show } f \neq \text{mbot} \land \text{sum}_2 M f \leq \text{sum}_2 M g \rightarrow h \oplus_M \text{sum}_2 M f \leq h \oplus_M \text{sum}_2 M g \]

proof
  assume 1: \[ f \neq \text{mbot} \land \text{sum}_2 M f \leq \text{sum}_2 M g \]
  hence \[ \exists k \, l \cdot f (k,l) \neq \text{bot} \]
  by (meson \text{agg-matrix-bot})
  hence \[ \textstyle \forall k \, \sum_i f (k,l) \neq \text{bot} \]
  using agg-sum-not-bot by blast
  obtain \[ c :: 'a \text{ where } \text{True} \]
  by simp
  have \( \textstyle (\sum_k \textstyle \sum_i f (k,l)) = (\text{sum}_2 M f) (c,c) \)
  by (simp add: \text{sum-matrix-2-def})
  also have \( \ldots \leq (\text{sum}_2 M g) (c,c) \)
  using 1 by (simp add: \text{less-eq-matrix-def})
  also have \( \ldots = (\sum_k \textstyle \sum_i g (k,l)) \)
  by (simp add: \text{sum-matrix-2-def})
  finally have \( \textstyle (\sum_k \textstyle \sum_i f (k,l)) \leq (\sum_k \textstyle \sum_i g (k,l)) \)
  by simp
  hence \( \textstyle (\sum_k \textstyle \sum_i f (k,l)) + (\sum_k \textstyle \sum_i g (k,l)) = (\text{sum}_2 M f) (c,c) \)
  by (metis \text{agg-sum-not-bot} \text{agg-matrix-bot})
  show \( h \oplus_M \text{sum}_2 M f \leq h \oplus_M \text{sum}_2 M g \)
proof (unfold \text{less-eq-matrix-def}, rule allI, unfold \text{plus-matrix-def})
  fix \( e \)
  have \( h \, e + (\text{sum}_2 M f) \, e = h \, e + (\sum_k \textstyle \sum_i f (k,l)) \)
  by (simp add: \text{sum-matrix-2-def})
  also have \( \ldots \leq h \, e + (\sum_k \textstyle \sum_i g (k,l)) \)
  using 2 3 by (metis \text{agg-sum-not-bot} \text{add-right-isotone} \text{add-commute})
  finally show \( h \, e + (\text{sum}_2 M f) \, e \leq h \, e + (\text{sum}_2 M g) \, e \)
  by (simp add: \text{sum-matrix-2-def})
qed

next

fix \f :: (‘a,’b) \text{ square}
show \text{sum}_2 M f \oplus_M \text{sum}_2 M mbot = \text{sum}_2 M f
by (simp add: \text{sum-bot-2 sum-plus-zero-2})

next

fix \f \, g :: (‘a,’b) \text{ square}
show \text{sum}_2 M f \oplus_M \text{sum}_2 M g = \text{sum}_2 M (f \oplus g) \oplus_M \text{sum}_2 M (f \otimes g)
proof
  fix \( e \)
  have \( (\text{sum}_2 M f \oplus_M \text{sum}_2 M g) \, e = (\text{sum}_2 M f) \, e + (\text{sum}_2 M g) \, e \)
  by (simp add: \text{plus-matrix-def})
  also have \( \ldots = (\sum_k \textstyle \sum_i f (k,l)) + (\sum_k \textstyle \sum_i g (k,l)) \)
  by (simp add: \text{sum-matrix-2-def})
  also have \( \ldots = (\sum_k \textstyle \sum_i f (k,l) + g (k,l)) \)
  using \text{agg-sum-distrib-2} by (metis \text{no-types})
  also have \( \ldots = (\sum_k \textstyle \sum_i (f (k,l) \cup g (k,l)) + (f (k,l) \cap g (k,l))) \)
using add-lattice aggregation_sum-0_cong by (metis (no-types, lifting))
also have ... = (∑ₖ ∑ₗ (f ⊕ g) (k,l)) + (∑ₖ ∑ₗ (f ⊗ g) (k,l))
by (simp add: sup-matrix-def inf-matrix-def)
also have ... = (∑ₖ ∑ₗ (f ⊕ g) (k,l)) + (∑ₖ ∑ₗ (f ⊗ g) (k,l))
using agg-sum-distrib-2 by (metis (no-types))
also have ... = (sum₂_M (f ⊗ g)) e + (sum₂_M (f ⊗ g)) e
by (simp add: sum-matrix-2-def)
also have ... = (sum₂_M (f ⊗ g) ⊕ₘ sum₂_M (f ⊗ g)) e
by (simp add: plus-matrix-def)
finally show (sum₂_M (f ⊗ₘ g)) e = (sum₂_M (f ⊗ g) ⊕ₘ sum₂_M (f ⊗ g)) e
by (simp add: sum-matrix-2-def)
also have ...
by (metis agg-sum-commute)
also have ...
by (simp add: sum-matrix-2-def)
finally show (sum₂_M (f ⊗ g)) e = (sum₂_M (f ⊗ g) ⊕ₘ sum₂_M (f ⊗ g)) e
by (simp add: plus-matrix-def)
finally show (sum₂_M (f ⊗ₘ g)) e = (sum₂_M (f ⊗ g) ⊕ₘ sum₂_M (f ⊗ g)) e
by (simp add: sum-matrix-2-def)
qed

next
fix f :: (′a,′b) square
show sum₂_M (fᵗ) = sum₂_M f
proof
fix e
have (sum₂_M (fᵗ)) e = (∑ₖ ∑ₗ (fᵗ) (k,l))
by (simp add: sum-matrix-2-def)
also have ...
by (simp add: conv-matrix-def)
also have ...
by simp
also have ...
by (metis agg-sum-commute)
also have ...
by (simp add: sum-matrix-2-def)
finally show (sum₂_M (fᵗ)) e = (sum₂_M f) e
by (simp add: sum-matrix-2-def)
qed

4.4 Matrix Minimisation

We construct an operation that finds the minimum entry of a matrix. Because a matrix can have several entries with the same minimum value, we introduce a lexicographic order on the indices to make the operation deterministic. The order is obtained by enumerating the universe of the index.

primrec enum-pos :: 'a list ⇒ 'a::enum ⇒ nat where
enum-pos Nil x = 0
| enum-pos (y#xs) x = (if x = y then 0 else 1 + enum-pos ys x)

lemma enum-pos_inverse: List.member xs x ⇒ xs!(enum-pos xs x) = x
apply (induct xs)
apply (simp add: member-rec(2))
by (metis diff-add-inverse enum-pos`.simps(2) less-one member-rec(1)
not-add-lessI nth-Cons)
The following function finds the position of an index in the enumerated universe.

fun enum-pos :: 'a::enum ⇒ nat where enum-pos x = enum-pos' (enum-class enum::'a list) x

lemma enum-pos-inverse [simp]:
  enum-class enum!(enum-pos x) = x
  apply (unfold enum-pos.simps)
  apply (rule enum-pos' inverse)
  by (metis in-enum List.member-def)

lemma enum-pos-injective [simp]:
  enum-pos x = enum-pos y =⇒ x = y
  by (metis enum-pos-inverse)

The position in the enumerated universe determines the order.

abbreviation enum-pos-less-eq :: 'a::enum ⇒ 'a⇒ bool where enum-pos-less-eq x y ≡ enum-pos-less i k ∨ (i = k ∧ enum-pos-less-eq j l)

abbreviation enum-pos-less :: 'a::enum ⇒ 'a⇒ bool where enum-pos-less x y ≡ enum-pos x < enum-pos y

Based on this, a lexicographic order is defined on pairs, which represent locations in a matrix.

abbreviation enum-lex-less :: 'a::enum × 'a ⇒ 'a×'a ⇒ bool where enum-lex-less ≡ λ(i,j) (k,l). enum-pos-less i k ∨ (i = k ∧ enum-pos-less-eq j l))

abbreviation enum-lex-less-eq :: 'a::enum × 'a ⇒ 'a×'a ⇒ bool where enum-lex-less-eq ≡ λ(i,j) (k,l). enum-pos-less i k ∨ (i = k ∧ enum-pos-less-eq j l))

The \( m \)-operation determines the location of the non-\( \bot \) minimum element which is first in the lexicographic order. The result is returned as a regular matrix with \( \top \) at that location and \( \bot \) everywhere else. In the weighted-graph model, this represents a single unweighted edge of the graph.

definition minarc-matrix :: ('a::enum,'b::{bot,ord,plus,top}) square ⇒ ('a,'b) square (minarc-M [80] 80) where minarc-matrix f = (λe . if f e = bot ∧ (∀ d . (f d ≠ bot → (f e + bot ≤ f d + bot ∧ (enum-lex-less d e → f e + bot ≠ f d + bot)))) then top else bot)

lemma minarc-at-most-one:
  fixes f :: ('a::enum,'b::{aggregation-order,top}) square
  assumes (minarc_M f) e = bot
  and (minarc_M f) d = bot
  shows e = d
proof –
  have 1: \( f e + \text{bot} \leq f d + \text{bot} \)
    by (metis assms minarc-matrix-def)
  have \( f d + \text{bot} \leq f e + \text{bot} \)
    by (metis assms minarc-matrix-def)
  hence \( f e + \text{bot} = f d + \text{bot} \)
    using 1 by simp
  hence \( \neg \text{enum-lex-less } d e \land \neg \text{enum-lex-less } e d \)
    using assms by (unfold minarc-matrix-def) (metis (lifting))
  thus \( ?\text{thesis} \)
    using enum-pos-injective less-linear by auto
qed

4.5 Linear Aggregation Lattices

We now assume that the aggregation order is linear and forms a bounded lattice. It follows that these structures are aggregation lattices. A linear order on matrix entries is necessary to obtain a unique minimum entry.

class linear-aggregation-lattice = linear-bounded-lattice + aggregation-order
begin

subclass aggregation-lattice
  apply unfold-locales
  by (metis add-commute sup-inf-selective)

sublocale heyting: bounded-heyting-lattice where implies = \( \lambda x y . \) if \( x \leq y \) then top else y
  apply unfold-locales
  by (simp add: inf-less-eq)
end

Every non-empty set with a transitive total relation has a least element with respect to this relation.

lemma least-order:
  assumes transitive: \( \forall x y z . \) le \( x y \land \) le \( y z \longrightarrow \) le \( x z \)
    and total: \( \forall x y . \) le \( x y \lor \) le \( y x \)
  shows finite \( A \implies A \neq \{} \implies \exists x . x \in A \land (\forall y . y \in A \longrightarrow \) le \( x y \)
proof (induct \( A \) rule: finite-ne-induct)
  case singleton
  thus \( ?\text{case} \)
    using total by auto
next
  case insert
  thus \( ?\text{case} \)
    by (metis insert_iff transitive total)
qed

lemma minarc-at-least-one:
fixes $f :: ('a::enum,'b::linear-aggregation-lattice) square$
assumes $f \neq mbot$
shows $\exists e . (minarc_M f) e = top$

proof —
let $?nbe = \{ (e,f) | e . f \neq bot \}$

have 1: finite $?nbe
  using finite-code finite-image-set by blast

have 2: $?nbe \neq {}$
  using assms agg-matrix-bot by fastforce

let $?le = \lambda (e::'a \times 'a,f::'b) (d::'a \times 'a,fd) . fe + bot \leq fd + bot$

have 3: $\forall x y z \ . \ ?le x y \land ?le y z \rightarrow ?le x z$
  by auto

have 4: $\forall x y \ . \ ?le x y \lor ?le y x$
  by (simp add: linear)

have $\exists x . x \in ?nbe \land (\forall y . y \in ?nbe \rightarrow ?le x y)$
  by (rule least-order, rule 3, rule 4, rule 1, rule 2)

then obtain $e fe$ where 5: $(e,fe) \in ?nbe \land (\forall y . y \in ?nbe \rightarrow ?le (e,fe) y)$
  by auto

let $?me = \{ e . f e \neq bot \land f e + bot = fe + bot \}$

have 6: finite $?me$
  using finite-code finite-image-set by blast

have 7: $?me \neq {}$
  using 5 by auto

have 8: $\forall x y z \ . \ enum-lex-less-eq x y \land enum-lex-less-eq y z \rightarrow enum-lex-less-eq x z$
  by auto

have 9: $\forall x y \ . \ enum-lex-less-eq x y \lor enum-lex-less-eq y x$
  by auto

have $\exists x . x \in ?me \land (\forall y . y \in ?me \rightarrow enum-lex-less-eq x y)$
  by (rule least-order, rule 8, rule 9, rule 6, rule 7)

then obtain $m$ where 10: $m \in ?me \land (\forall y . y \in ?me \rightarrow enum-lex-less-eq m y)$
  by auto

have 11: $f m \neq bot$
  using 10 5 by auto

have 12: $\forall d . f d \neq bot \rightarrow f m + bot \leq fd + bot$
  using 10 5 by simp

have $\forall d . f d \neq bot \land enum-lex-less d m \rightarrow f m + bot \neq fd + bot$
  using 10 by fastforce

hence $(minarc_M f) m = top$
  using 11 12 by (simp add: minarc-matrix-def)
thus $?thesis$
  by blast

qed

Linear aggregation lattices form a Stone relation algebra by reusing the meet operation as composition, using identity as converse and a standard implementation of pseudocomplement.

class linear-aggregation-algebra = linear-aggregation-lattice + uminus + one +
\begin{verbatim}
times + conv + assumes uminus-def-2 [simp]: \(-x = (\text{if } x = \text{bot} \text{ then } \text{top} \text{ else } \text{bot})\)
asumes one-def-2 [simp]: \(1 = \text{top}\)
asumes times-def-2 [simp]: \(x \ast y = x \land y\)
asumes conv-def-2 [simp]: \(x^T = x\)
begin
subclass aggregation-algebra apply unfold-locales using inf-dense by auto

lemma regular-bot-top-2: regular x \iff x = \text{bot} \lor x = \text{top}
by simp

sublocale heyting: heyting-stone-algebra where implies = \(\lambda x y. \text{if } x \leq y \text{ then } \text{top else } y\)
apply unfold-locales
apply (simp add: antisym)
by auto

end

We show that matrices over linear aggregation lattices form an m-algebra using the above operations.
interpretation agg-square-m-algebra: \text{m-algebra} where sup = sup-matrix and
inf = inf-matrix and less-eq = less-eq-matrix and less = less-matrix and bot =
bot-matrix::('a::enum, 'b::linear-aggregation-algebra) square and top = top-matrix
and uminus = uminus-matrix and one = one-matrix and times = times-matrix
and conv = conv-matrix and plus = plus-matrix and sum = sum-matrix and
minarc = minarc-matrix
proof
fix f :: ('a,'b) square
show \text{minarc}\_\text{M} f \leq \ominus\ominus f
proof (unfold less-eq-matrix-def, rule allI)
fix e :: 'a \times 'a
have (minarc\_\text{M} f) e \leq (if f e \neq \text{bot} \text{ then } \text{top else } \ominus(f e))
by (simp add: minarc-matrix-def)
also have \ldots = \ominus(f e)
by simp
also have \ldots = (\ominus f) e
by (simp add: uminus-matrix-def)
finally show (minarc\_\text{M} f) e \leq (\ominus f) e
.qed
next
fix f :: ('a,'b) square
let \text{at} = \text{bounded-distrib-allegory-signature.arc} mone times-matrix
less-eq-matrix mtop conv-matrix

\end{verbatim}
show \( f \neq \text{mbot} \rightarrow \exists t \in (\text{minarc}_M f) \)
proof
  assume 1: \( f \neq \text{mbot} \)
  have \( \text{minarc}_M f \odot \text{mtop} \odot (\text{minarc}_M f \odot \text{mtop})^T = \text{minarc}_M f \odot \text{mtop} \odot (\text{minarc}_M f)^T \)
  by (metis matrix-bounded-idempotent-semiring.surjective-top-closed
       matrix-monoid.mult-assoc matrix-stone-relation-algebra.conv-dist-comp
       matrix-stone-relation-algebra.conv-top)
  also have ... \(\leq \text{mone} \)
  proof (unfold less-eq-matrix-def, rule allI, rule prod-cases)
    fix \( i \ j \)
    have \((\text{minarc}_M f \odot \text{mtop} \odot (\text{minarc}_M f)^T) (i,j) = (\bigcup_k (\text{minarc}_M f) (i,k) * \text{mtop} (k,l)) * ((\text{minarc}_M f)^T) (l,j)) \)
    by (simp add: times-matrix-def)
    also have ... \(= (\bigcup_k (\text{minarc}_M f) (i,k) * \text{top} * ((\text{minarc}_M f)^T) (j,l)) \)
    by (simp add: top-matrix-def conv-matrix-def)
    also have ... \(= (\bigcup_k \text{mone} (\text{minarc}_M f) (i,k) * \text{top} * ((\text{minarc}_M f)^T) (j,l)) \)
    by (metis comp-right-dist-sum)
    also have ... \(= (\bigcup_k \text{mone} if i = j \wedge l = k then (\text{minarc}_M f) (i,k) * \text{top} * ((\text{minarc}_M f) (j,l))^T else bot) \)
    by auto
    also have ... \(\leq (if i = j then \text{top} else bot) \)
    by simp
    also have ... \(= \text{mone} (i,j) \)
    by (simp add: one-matrix-def)
    finally show \((\text{minarc}_M f \odot \text{mtop} \odot (\text{minarc}_M f)^T) (i,j) \leq \text{mone} (i,j) \)
  .
  qed
finally have 2: \(\text{minarc}_M f \odot \text{mtop} \odot (\text{minarc}_M f \odot \text{mtop})^T \leq \text{mone} \)
. have 3: \(\text{mtop} \odot (\text{minarc}_M f \odot \text{mtop}) = \text{mtop} \)
proof (rule ext, rule prod-cases)
  fix \( i \ j \)
  from \( \text{minarc-at-least-one} \) obtain \( e_i \ \text{ej} \) where \((\text{minarc}_M f) (e_i, ej) = \text{top} \)
  using 1 by force
  hence 4: \( \text{top} \odot \text{top} \leq (\bigcup_l (\text{minarc}_M f) (e_i, l) \odot \text{top}) \)
  by (metis comp-inf.unb-sum)
  have \( \text{top} \odot (\bigcup_l (\text{minarc}_M f) (e_i, l) \odot \text{top}) \leq (\bigcup_k \text{top} \odot (\bigcup_l (\text{minarc}_M f) (k,l) \odot \text{top})) \)
  by (rule comp-inf.unb-sum)
  hence \( \text{top} \leq (\bigcup_k \text{top} \odot (\bigcup_l (\text{minarc}_M f) (k,l) \odot \text{top})) \)
  using 4 by auto
  also have ... \(= (\bigcup_k \text{mtop} (i,k) * (\bigcup_l (\text{minarc}_M f) (k,l) * \text{mtop} (l,j))) \)
. 38
by (simp add: top-matrix-def)
also have ... = (mtop ⊙ (minarc_M f ⊙ mtop)) (i,j)
by (simp add: times-matrix-def)
finally show (mtop ⊙ (minarc_M f ⊙ mtop)) (i,j) = mtop (i,j)
by (simp add: eq-iff top-matrix-def)
qed

have (minarc_M f)^t ⊙ mtop ⊙ ((minarc_M f)^t ⊙ mtop)^t = (minarc_M f)^t ⊙ mtop ⊙ (minarc_M f)
by (metis matrix-stone-relation-algebra.comp-associative
matrix-stone-relation-algebra.conv-dist-comp
matrix-stone-relation-algebra.conv-involutive
matrix-stone-relation-algebra.conv-top
matrix-bounded-idempotent-semiring.convjective-top-closed)
also have ... ≤ mone
proof (unfold less-eq-matrix-def, rule allI, rule prod-cases)
fix i j
have ((minarc_M f)^t ⊙ mtop ⊙ minarc_M f) (i,j) = (∪_k ((minarc_M f)^t) (i,k) * mtop (k,l)) * (minarc_M f) (l,j)
by (simp add: times-matrix-def)
also have ... = (∪_k ((minarc_M f) (k,i))^T * top) * (minarc_M f) (l,j)
by (simp add: top-matrix-def conv-matrix-def)
also have ... = (∪_k ((minarc_M f) (k,i))^T * top) * (minarc_M f) (l,j)
by (metis comp-right-dist-sum)
also have ... = (∪_k if i = j ∧ l = k then ((minarc_M f) (k,i))^T * top *
(minarc_M f) (l,j) else bot)
apply (rule sap-monoid.sum.cong)
apply simp
by (metis (no-types, lifting) comp-left-zero comp-right-zero conv-bot
prod.inject minarc-at-most-one)
also have ... = (if i = j then (∪_k if l = k then ((minarc_M f) (k,i))^T *
top * (minarc_M f) (l,j) else bot) else bot)
by auto
also have ... ≤ (if i = j then top else bot)
by simp
also have ... = mone (i,j)
by (simp add: one-matrix-def)
finally show ((minarc_M f)^t ⊙ mtop ⊙ (minarc_M f)) (i,j) ≤ mone (i,j)
qed

finally have δ: (minarc_M f)^t ⊙ mtop ⊙ ((minarc_M f)^t ⊙ mtop)^t ≤ mone

have mtop ⊙ ((minarc_M f)^t ⊙ mtop) = mtop
using 3 by (metis matrix-monoid.mult-assoc
matrix-stone-relation-algebra.conv-dist-comp
matrix-stone-relation-algebra.conv-top)
thus ?at (minarc_M f)
using 2 3 5 by blast
qed

next
fix $f, g :: ('a, 'b) square$

let $?at = bounded-distrib-allegory-signature. arc mone times-matrix
less-eq-matrix mtop conv-matrix$

show $?at g \land g \otimes f \neq \mbot \longrightarrow \sum_M (\minarc_M f \otimes f) \leq \sum_M (g \otimes f)$

proof
assume 1: $?at g \land g \otimes f \neq \mbot$

hence 2: $g = \ominus g$

using matrix-stone-relation-algebra. arc-regular by blast

show $\sum_M (\minarc_M f \otimes f) \leq \sum_M (g \otimes f)$

proof (unfold less-eq-matrix-def, rule allI, rule prod-cases)
fix $i, j$

from minarc-at-least-one obtain $ei, ej$ where 3: $(\minarc_M f) (ei, ej) = \top$

using 1 by force

hence 4: $\forall k, l. \neg (k = ei \land l = ej) \longrightarrow (\minarc_M f) (k, l) = \bot$

by (metis (mono-tags, hide-lams) bot.extremum inf. bot-unique prod.inject
\minarc-at-most-one)

from agg-matrix-bot obtain $di, dj$ where 5: $(g \otimes f) (di, dj) \neq \bot$

using 1 by force

hence 6: $g (di, dj) \neq \bot$

by (metis inf-bot-left inf-matrix-def)

hence 7: $g (di, dj) = \top$

using 2 by (metis muminus-matrix-def muminus-def)

hence 8: $(g \otimes f) (di, dj) = f (di, dj)$

by (metis inf-matrix-def inf-top.left-neutral)

have 9: $\forall k, l. k \neq di \longrightarrow g (k, l) = \bot$

proof (intro allI, rule impI)
fix $k, l$

assume 10: $k \neq di$

have $top * (g (k, l))^T = g (di, dj) * top * (g^t) (l, k)$

using 7 by (simp add: conv-matrix-def)

also have $\ldots \leq (\bigcup_n g (di, n) * top) * (g^t) (l, k)$

by (metis comp-inf.ub-sum comp-right-dist-sum)

also have $\ldots \leq (\bigcup_n (\bigcup_n g (di, n) * top) * (g^t) (m, k))$

by (metis comp-inf.ub-sum)

also have $\ldots = (g \otimes mtop \otimes g^t) (di, k)$

by (simp add: times-matrix-def top-matrix-def)

also have $\ldots \leq mone (di, k)$

using 1 by (metis matrix-stone-relation-algebra. arc-expanded
\less-eq-matrix-def)

also have $\ldots = \bot$

apply (unfold one-matrix-def)

using 10 by auto

finally have $g (k, l) \neq \top$

using 5 by (metis bot.extremum conv-def inf. bot-unique mult.left-neutral
\one-def)

thus $g (k, l) = \bot$

using 2 by (metis muminus-def muminus-matrix-def)

qed

have $\forall k, l. l \neq dj \longrightarrow g (k, l) = \bot$

40
proof (intro allI, rule impI)
fix k l
assume 11: l ≠ dj
have (g (k,l))T * top = (g') (l,k) * top * g (dij)
  using 7 by (simp add: comp-associative cone-matrix-def)
also have ... ≤ (∏ₙ (g') (l,n) * top) * g (dij)
  by (metis comp-inf.ub-sum comp-right-dist-sum)
also have ... ≤ (∏ₙ (γ (l,n) * top) * g (m,dj))
  by (metis comp-inf.ub-sum)
also have ... = (g' ⊗ g) (l,dj)
  by (simp add: times-matrix-def top-matrix-def)
also have ... ≤ mone (l,dj)
  using 1 by (metis matrix-stone-relation-algebra.arc-expanded
less-eq-matrix-def)
also have ... = bot
  apply (unfold one-matrix-def)
  using 11 by auto
finally have g (k,l) ≠ top
  using 5 by (metis bot.extremum comp-right-one conv-def one-def
top.extremum-unique)
thus g (k,l) = bot
  using 2 by (metis uminus-def uminus-matrix-def)
qed

hence 12: ∀ k l . ¬(k = di ∧ l = dj) → (g ⊗ f) (k,l) = bot
  using 9 by (metis inf-bot-left inf-matrix-def)
have (∑ₖ ∑ₖ (minarcₘ f ⊗ f) (k,l)) = (∑ₖ ∑ₖ if k = ei ∧ l = ej then
  (minarcₘ f) (k,l) else (minarcₘ f ⊗ f) (k,l))
  by simp
also have ... = (∑ₖ ∑ₖ if k = ei ∧ l = ej then (minarcₘ f) (k,l) else
  (minarcₘ f) (k,l) ∩ f (k,l))
  by (unfold inf-matrix-def) simp
also have ... = (∑ₖ ∑ₖ if k = ei ∧ l = ej then (minarcₘ f ⊗ f) (k,l) else
  bot)
  apply (rule aggregation.sum-0.cong)
  apply simp
  using 4 by (metis inf-bot-left)
also have ... = (minarcₘ f ⊗ f) (ei,ej) + bot
  by (unfold agg-delta-2) simp
also have ... = f (ei,ej) + bot
  using 3 by (simp add: inf-matrix-def)
also have ... ≤ (g ⊗ f) (di,dj) + bot
  using 3 5 6 7 8 by (metis minarc-matrix-def)
also have ... = (∑ₖ ∑ₖ if k = di ∧ l = dj then (g ⊗ f) (k,l) else bot)
  by (unfold agg-delta-2) simp
also have ... = (∑ₖ ∑ₖ if k = di ∧ l = dj then (g ⊗ f) (k,l) else (g ⊗ f)
  (k,l))
  apply (rule aggregation.sum-0.cong)
  apply simp
  using 12 by metis
also have ... = \( \sum_k \sum_l (g \otimes f) (k,l) \)

by simp

finally show \( (\text{sum}_M (\text{minarc}_M f \otimes f)) (i,j) \leq (\text{sum}_M (g \otimes f)) (i,j) \)

by (simp add: sum-matrix-def)

qed

qed

next

fix \( f, g :: ('a,'b) \text{ square} \)

let \(?h = \text{hd enum-class.enum} \)

show \( \text{sum}_M f \leq \text{sum}_M g \lor \text{sum}_M g \leq \text{sum}_M f \)

proof (cases \( (\text{sum}_M f) (?h,?h) \leq (\text{sum}_M g) (?h,?h) \))

  case 1: True

  have \( \text{sum}_M f \leq \text{sum}_M g \)

  apply (unfold less-eq-matrix-def, rule allI, rule prod-cases)

  thus \(?\text{thesis} \)

  by simp

next

  case False

  hence \( 2: (\text{sum}_M g) (?h,?h) \leq (\text{sum}_M f) (?h,?h) \)

  by (simp add: linear)

  have \( \text{sum}_M g \leq \text{sum}_M f \)

  apply (unfold less-eq-matrix-def, rule allI, rule prod-cases)

  using \( 2 \) by (unfold sum-matrix-def) auto

  thus \(?\text{thesis} \)

  by simp

qed

next

  have finite \( \{ f :: ('a,'b) \text{ square} . (\forall e . \text{regular} (f e)) \} \)

  by (unfold regular-bot-top-2, rule finite-set-of-finite-funs-pred) auto

  thus finite \( \{ f :: ('a,'b) \text{ square} . \text{matrix-p-algebra.regular} f \} \)

  by (unfold uminus-matrix-def) meson

qed

We show the same for the alternative implementation that stores the result of aggregation in all elements of the matrix.

interpretation agg-square-m-algebra-2: \( \text{m-algebra} \) where \( \text{sup} = \text{sup-matrix} \) and \( \text{inf} = \text{inf-matrix} \) and less-eq = less-eq-matrix and less = less-matrix and bot = bot-matrix::\( ('a::\text{enum},'b::\text{linear-aggregation-algebra}) \text{ square} \) and top = top-matrix and uminus = uminus-matrix and one = one-matrix and times = times-matrix and conv = conv-matrix and plus = plus-matrix and sum = sum-matrix-2 and minarc = minarc-matrix

proof

fix \( f :: ('a,'b) \text{ square} \)

show \( \text{minarc}_M f \leq \emptyset f \)

by (simp add: agg-square-m-algebra.minarc-below)

next

fix \( f :: ('a,'b) \text{ square} \)

let \(?\text{at = bounded-distrib-allegory-signature.arc mone times-matrix} \)

42
By defining the Kleene star as $\top$ aggregation lattices form a Kleene algebra.

class aggregation-kleene-algebra = aggregation-algebra + star +
  assumes star-def [simp]: $x^* = \top$
begin
subclass stone-kleene-relation-algebra
  apply unfold-locales
  by (simp-all add: inf.assoc le-infI2 inf-sup-distrib1 inf-sup-distrib2)
end

class linear-aggregation-kleene-algebra = linear-aggregation-algebra + star +
  assumes star-def-2 [simp]: $x^* = \top$
begin

subclass aggregation-kleene-algebra
  apply unfold-locales
  by simp
end

interpretation agg-square-m-kleene-algebra: m-kleene-algebra where sup = sup-matrix and inf = inf-matrix and less-eq = less-eq-matrix and less = less-matrix and bot = bot-matrix::('a::enum,'b::linear-aggregation-kleene-algebra) square and top = top-matrix and uminus = uminus-matrix and one = one-matrix and times = times-matrix and conv = conv-matrix and star = star-matrix and plus = plus-matrix and sum = sum-matrix and minarc = minarc-matrix ..

interpretation agg-square-m-kleene-algebra-2: m-kleene-algebra where sup = sup-matrix and inf = inf-matrix and less-eq = less-eq-matrix and less = less-matrix and bot = bot-matrix::('a::enum,'b::linear-aggregation-kleene-algebra) square and top = top-matrix and uminus = uminus-matrix and one = one-matrix and times = times-matrix and conv = conv-matrix and star = star-matrix and plus = plus-matrix and sum = sum-matrix-2 and minarc = minarc-matrix ..

end

5 Algebras for Aggregation and Minimisation with a Linear Order

This theory gives several classes of instances of linear aggregation lattices as described in [4]. Each of these instances can be used as edge weights and the resulting graphs will form s-algebras and m-algebras as shown in a separate theory.

theory Linear-Aggregation-Algebras

imports Matrix-Aggregation-Algebras HOL.Real

begin

no-notation
inf (infixl ⊓ 70)
and uminus (¬ - [81] 80)

5.1 Linearly Ordered Commutative Semigroups

Any linearly ordered commutative semigroup extended by new least and greatest elements forms a linear aggregation lattice. The extension is done so that the new least element is a unit of aggregation and the new greatest element is a zero of aggregation.
datatype 'a ext =
    Bot
  | Val 'a
  | Top

instantiation ext :: (linordered-ab-semigroup-add)
linear-aggregation-kleene-algebra
begin

fun plus-ext :: 'a ext ⇒ 'a ext ⇒ 'a ext where
    plus-ext Bot x = x
  | plus-ext (Val x) Bot = Val x
  | plus-ext (Val x) (Val y) = Val (x + y)
  | plus-ext (Val -) Top = Top
  | plus-ext Top - = Top

fun sup-ext :: 'a ext ⇒ 'a ext ⇒ 'a ext where
    sup-ext Bot x = x
  | sup-ext (Val x) Bot = Val x
  | sup-ext (Val x) (Val y) = Val (max x y)
  | sup-ext (Val -) Top = Top
  | sup-ext Top - = Top

fun inf-ext :: 'a ext ⇒ 'a ext ⇒ 'a ext where
    inf-ext Bot - = Bot
  | inf-ext (Val -) Bot = Bot
  | inf-ext (Val x) (Val y) = Val (min x y)
  | inf-ext (Val x) Top = Val x
  | inf-ext Top x = x

fun times-ext :: 'a ext ⇒ 'a ext ⇒ 'a ext where
    times-ext x y = x \cap y

fun uminus-ext :: 'a ext ⇒ 'a ext where
    uminus-ext Bot = Top
  | uminus-ext (Val -) = Bot
  | uminus-ext Top = Bot

fun star-ext :: 'a ext ⇒ 'a ext where
    star-ext - = Top

fun conv-ext :: 'a ext ⇒ 'a ext where
    conv-ext x = x

definition bot-ext :: 'a ext where
    bot-ext = Bot

definition one-ext :: 'a ext where
    one-ext = Top

definition top-ext :: 'a ext where
    top-ext = Top

fun less-eq-ext :: 'a ext ⇒ 'a ext ⇒ bool where
    less-eq-ext Bot - = True
  | less-eq-ext (Val -) Bot = False
  | less-eq-ext (Val x) (Val y) = (x ≤ y)

45
fun less-ext :: 'a ext ⇒ 'a ext ⇒ bool where less-ext x y = (x ≤ y ∧ ¬ y ≤ x)

instance

proof

  fix x y z :: 'a ext
  show (x + y) + z = x + (y + z)
    by (cases x; cases y; cases z) (simp-all: addassoc)
  show x + y = y + x
    by (cases x; cases y) (simp-all: addcommute)
  show (x < y) = (x ≤ y ∧ ¬ y ≤ x)
    by simp
  show x ≤ x
    using less-eq-ext.elims(3) by fastforce
  show x ≤ y ⇒ y ≤ z ⇒ x ≤ z
    by (cases x; cases y; cases z) simp-all
  show x ≤ y ⇒ y ≤ x ⇒ x = y
    by (cases x; cases y) simp-all
  show x ∩ y ≤ x
    by (cases x; cases y) simp-all
  show x ⊓ y ≤ y
    by (cases x; cases y) simp-all
  show x ≤ x ⊔ y
    by (cases x; cases y) simp-all
  show y ≤ x ⊔ y
    by (cases x; cases y) simp-all
  show y ≤ x ⇒ z ≤ x ⇒ y ⊔ z ≤ x
    by (cases x; cases y; cases z) simp-all
  show bot ≤ x
    by (simp add: bot-ext-def)
  show x ≤ top
    by (cases x) (simp-all add: top-ext-def)
  show x ≠ bot ∧ x + bot ≤ y + bot → x + z ≤ y + z
    by (cases x; cases y; cases z) (simp-all add: bot-ext-def add-right-mono)
  show x + y + bot = x + y
    by (cases x; cases y) (simp-all add: bot-ext-def)
  show x + y = bot → x = bot
    by (cases x; cases y) (simp-all add: bot-ext-def)
  show x ≤ y ∨ y ≤ x
    by (cases x; cases y) (simp-all add: bot-ext-def)
  show ¬x = (if x = bot then top else bot)
    by (cases x) (simp-all add: bot-ext-def top-ext-def)
  show (1::'a ext) = top

46
by (simp add: one-ext-def top-ext-def)
show \( x \ast y = x \sqcap y \)
  by simp
show \( x^{T} = x \)
  by simp
show \( x^{*} = \top \)
  by (simp add: top-ext-def)
qed

end

An example of a linearly ordered commutative semigroup is the set of real numbers with standard addition as aggregation.

lemma example-real-ext-matrix:
  fixes \( x :: (\prime a :: \text{enum}, \text{real ext}) \) square
  shows minarc \( M \seq x \sqsubseteq \sqsubseteq x \)
  by (rule agg-square-m-algebra.minarc-below)

Another example of a linearly ordered commutative semigroup is the set of real numbers with maximum as aggregation.

datatype real-max = Rmax real
instantiation real-max :: linordered-ab-semigroup-add
begin
  fun less-eq-real-max where
    less-eq-real-max (Rmax x) (Rmax y) = (x \leq y)
  fun less-real-max where
    less-real-max (Rmax x) (Rmax y) = (x < y)
  fun plus-real-max where
    plus-real-max (Rmax x) (Rmax y) = Rmax (max x y)

instance
proof
  fix \( x \ y \ z :: \text{real-max} \)
  show \( (x + y) + z = x + (y + z) \)
    by (cases x; cases y; cases z) simp
  show \( x + y = y + x \)
    by (cases x; cases y) simp
  show \( (x < y) = (x \leq y \land \neg y \leq x) \)
    by (cases x; cases y) auto
  show \( x \leq x \)
    by (cases x) simp
  show \( x \leq y \implies y \leq z \implies x \leq z \)
    by (cases x; cases y; cases z) simp
  show \( x \leq y \implies y \leq x \implies x = y \)
    by (cases x; cases y) simp
  show \( x \leq y \implies z + x \leq z + y \)
    by (cases x; cases y; cases z) simp
  show \( x \leq y \implies y \vee x \leq x \)
    by (cases x; cases y) auto
qed

47
A third example of a linearly ordered commutative semigroup is the set of real numbers with minimum as aggregation.

datatype real-min = Rmin real

instantiation real-min :: linordered-ab-semigroup-add

begin

fun less-eq-real-min where less-eq-real-min (Rmin x) (Rmin y) = (x ≤ y)
fun less-real-min where less-real-min (Rmin x) (Rmin y) = (x < y)
fun plus-real-min where plus-real-min (Rmin x) (Rmin y) = Rmin (min x y)

instance
proof
fix x y z :: real-min
show (x + y) + z = x + (y + z)
by (cases x; cases y; cases z) simp
show x + y = y + x
by (cases x; cases y) simp
show (x < y) = (x ≤ y ∧ ¬ y ≤ x)
by (cases x; cases y) auto
show x ≤ x
by (cases x) simp
show x ≤ y =⇒ y ≤ z =⇒ x ≤ z
by (cases x; cases y; cases z) simp
show x ≤ y =⇒ y ≤ x =⇒ x = y
by (cases x; cases y) simp
show x ≤ y =⇒ z + x ≤ z + y
by (cases x; cases y; cases z) simp
show x ≤ y ∨ y ≤ x
by (cases x; cases y) auto
qed

end

lemma example-real-min-ext-matrix:
fixes x :: ('a::enum,real-min ext) square
shows minarcM x ≤ ⊗x
by (rule agg-square-m-algebra.minarc-below)

5.2 Linearly Ordered Commutative Monoids
Any linearly ordered commutative monoid extended by new least and greatest elements forms a linear aggregation lattice. This is similar to linearly ordered commutative semigroups except that the aggregation \( \bot + \bot \) produces the unit of the monoid instead of the least element. Applied to weighted graphs, this means that the aggregation of the empty graph will be the unit of the monoid (for example, 0 for real numbers under standard addition, instead of \( \bot \)).

```plaintext
class linordered-comm-monoid-add = linordered-ab-semigroup-add + comm-monoid-add
datatype 'a ext0 |
  Bot |
  Val 'a |
  Top

instantiation ext0 :: (linordered-comm-monoid-add)
linear-aggregation-kleene-algebra
begin

fun plus-ext0 :: 'a ext0 \Rightarrow 'a ext0 \Rightarrow 'a ext0 where
  plus-ext0 Bot Bot = Val 0
| plus-ext0 Bot x = x
| plus-ext0 (Val x) Bot = Val x
| plus-ext0 (Val x) (Val y) = Val (x + y)
| plus-ext0 (Val -) Top = Top
| plus-ext0 Top - = Top

fun sup-ext0 :: 'a ext0 \Rightarrow 'a ext0 \Rightarrow 'a ext0 where
  sup-ext0 Bot x = x
| sup-ext0 (Val x) Bot = Val x
| sup-ext0 (Val x) (Val y) = Val (max x y)
| sup-ext0 (Val -) Top = Top
| sup-ext0 Top - = Top

fun inf-ext0 :: 'a ext0 \Rightarrow 'a ext0 \Rightarrow 'a ext0 where
  inf-ext0 Bot - = Bot
| inf-ext0 (Val -) Bot = Bot
| inf-ext0 (Val x) (Val y) = Val (min x y)
| inf-ext0 (Val x) Top = Val x
| inf-ext0 Top x = x

fun times-ext0 :: 'a ext0 \Rightarrow 'a ext0 \Rightarrow 'a ext0 where
  times-ext0 x y = x \sqcap y

fun uminus-ext0 :: 'a ext0 \Rightarrow 'a ext0 where
  uminus-ext0 Bot = Top
| uminus-ext0 (Val -) = Bot
| uminus-ext0 Top = Bot
```

49
fun star-ext0 :: 'a ext0 ⇒ 'a ext0 where star-ext0 - = Top

fun conv-ext0 :: 'a ext0 ⇒ 'a ext0 where conv-ext0 x = x

definition bot-ext0 :: 'a ext0 where bot-ext0 ≡ Bot

definition one-ext0 :: 'a ext0 where one-ext0 ≡ Top

definition top-ext0 :: 'a ext0 where top-ext0 ≡ Top

fun less-eq-ext0 :: 'a ext0 ⇒ 'a ext0 ⇒ bool where
less-eq-ext0 Bot - = True
| less-eq-ext0 (Val -) Bot = False
| less-eq-ext0 (Val x) (Val y) = (x ≤ y)
| less-eq-ext0 (Val -) Top = True
| less-eq-ext0 Top Bot = False
| less-eq-ext0 Top (Val -) = False
| less-eq-ext0 Top Top = True

fun less-ext0 :: 'a ext0 ⇒ 'a ext0 ⇒ bool where
less-ext0 x y = (x ≤ y ∧ ¬ y ≤ x)

instance
definition

proof

fix x y z :: 'a ext0

show (x + y) + z = x + (y + z)
  by (cases x; cases y; cases z) (simp-all add: add_assoc)

show x + y = y + x
  by (cases x; cases y) (simp-all add: add.commute)

show (x < y) = (x ≤ y ∧ ¬ y ≤ x)
  by simp

show x ≤ x
  using less-eq-ext0.elims(3) by fastforce

show x ≤ y =⇒ y ≤ z =⇒ x ≤ z
  by (cases x; cases y; cases z) simp-all

show x ≤ y =⇒ y ≤ x =⇒ x = y
  by (cases x; cases y) simp-all

show x ⊓ y ≤ x
  by (cases x; cases y) simp-all

show x ⊓ y ≤ y
  by (cases x; cases y) simp-all

show x ≤ y =⇒ z ≤ x =⇒ x ⊓ z ≤ y
  by (cases x; cases y; cases z) simp-all

show x ≤ x ⊔ y
  by (cases x; cases y) simp-all

show y ≤ x ⊔ y
  by (cases x; cases y) simp-all

show y ≤ x =⇒ z ≤ x =⇒ y ⊔ z ≤ x
  by (cases x; cases y; cases z) simp-all

show bot ≤ x
  by (simp add: bot-ext0-def)
show \( x \leq \text{top} \)
by (cases \( x \)) (simp-all add: top-ext0-def)

show \( x \neq \text{bot} \land x + \text{bot} \leq y + \text{bot} \rightarrow x + z \leq y + z \)
apply (cases \( x \); cases \( y \); cases \( z \))
prefer 11 using add-right-mono bot-ext0-def apply fastforce
by (simp-all add: bot-ext0-def add-right-mono)

show \( x + y + \text{bot} = x + y \)
by (cases \( x \); cases \( y \)) (simp-all add: bot-ext0-def)

show \( x + y = \text{bot} \rightarrow x = \text{bot} \)
by (cases \( x \); cases \( y \)) (simp-all add: bot-ext0-def)

show \( x \leq y \lor y \leq x \)
by (cases \( x \); cases \( y \)) (simp-all add: linear)

show \(-x = (if x = \text{bot} \text{ then top else bot})\)
by (cases \( x \)) (simp-all add: bot-ext0-def top-ext0-def)

show \((1::'a ext0) = \text{top}\)
by (simp add: one-ext0-def top-ext0-def)

show \( x * y = x \cap y \)
by simp

show \( x^T = x \)
by simp

show \( x^* = \text{top} \)
by (simp add: top-ext0-def)
qed

5.3 Linearly Ordered Commutative Monoids with a Least Element

If a linearly ordered commutative monoid already contains a least element which is a unit of aggregation, only a new greatest element has to be added to obtain a linear aggregation lattice.

class linordered-comm-monoid-add-bot = linordered-ab-semigroup-add + order-bot +
assumes bot-zero [simp]: \( \text{bot} + x = x \)
begin

sublocale linordered-comm-monoid-add where \( \text{zero} = \text{bot} \)
apply unfold-locales
by simp

end

datatype 'a extT =
  Val 'a
| Top

instantiation extT :: (linordered-comm-monoid-add-bot)
linear-aggregation-kleene-algebra
begin

fun plus-extT :: 'a extT ⇒ 'a extT ⇒ 'a extT where
  plus-extT (Val x) (Val y) = Val (x + y)
| plus-extT (Val -) Top = Top
| plus-extT Top - = Top

fun sup-extT :: 'a extT ⇒ 'a extT ⇒ 'a extT where
  sup-extT (Val x) (Val y) = Val (max x y)
| sup-extT (Val -) Top = Top
| sup-extT Top - = Top

fun inf-extT :: 'a extT ⇒ 'a extT ⇒ 'a extT where
  inf-extT (Val x) (Val y) = Val (min x y)
| inf-extT (Val x) Top = Val x
| inf-extT Top x = x

fun times-extT :: 'a extT ⇒ 'a extT ⇒ 'a extT where
  times-extT x y = x \odot y

fun uminus-extT :: 'a extT ⇒ 'a extT where
  uminus-extT x = (if x = Val bot then Top else Val bot)

fun star-extT :: 'a extT ⇒ 'a extT where
  star-extT - = Top

fun conv-extT :: 'a extT ⇒ 'a extT where
  conv-extT x = x

definition bot-extT :: 'a extT where
  bot-extT ≡ Val bot

definition one-extT :: 'a extT where
  one-extT ≡ Top

definition top-extT :: 'a extT where
  top-extT ≡ Top

fun less-eq-extT :: 'a extT ⇒ 'a extT ⇒ bool where
  less-eq-extT (Val x) (Val y) = (x ≤ y)
| less-eq-extT Top (Val -) = False
| less-eq-extT - Top = True

fun less-extT :: 'a extT ⇒ 'a extT ⇒ bool where
  less-extT x y = (x ≤ y ∧ ¬ y ≤ x)

instance
proof
fix x y z :: 'a extT
show \((x + y) + z = x + (y + z)\)
  by (cases x; cases y; cases z) (simp-all add: add.assoc)
show \(x + y = y + x\)
  by (cases x; cases y) (simp-all add: add.commute)
show \((x < y) = (x \leq y \land \neg y \leq x)\)
  by simp
show \(x \leq x\)
  by (cases x) simp-all
show \(x \leq y \Longrightarrow y \leq z \Longrightarrow x \leq z\)
  by (cases x; cases y; cases z) simp-all
show \(x \leq y \Longrightarrow y \leq x \Longrightarrow x = y\)
  by (cases x; cases y) simp-all
show \(x \cap y \leq x\)
  by (cases x; cases y) simp-all
show \(x \cap y \leq y\)
  by (cases x; cases y) simp-all
show \(x \leq y \Longrightarrow x \leq z \Longrightarrow x \leq y \cap z\)
  by (cases x; cases y; cases z) simp-all
show \(x \leq x \sqcup y\)
  by (cases x; cases y) simp-all
show \(y \leq x \sqcup y\)
  by (cases x; cases y) simp-all
show \(y \leq x \Longrightarrow z \leq x \Longrightarrow y \sqcup z \leq x\)
  by (cases x; cases y; cases z) simp-all
show \(bot \leq x\)
  by (cases x) (simp-all add: bot-extT-def)
show \(x \leq top\)
  by (cases x) (simp-all add: top-extT-def)
show \(x \neq bot \land x + bot \leq y + bot \longrightarrow x + z \leq y + z\)
  by (cases x; cases y; cases z) (simp-all add: bot-extT-def add-right-mono)
show \(x + y + bot = x + y\)
  by (cases x; cases y) (simp-all add: bot-extT-def)
show \(x + y = bot \longrightarrow x = bot\)
  apply (cases x; cases y)
    apply (metis (mono-tags) add.commute add-right-mono bot.extremum
      bot.extremum-uniqueI bot-zero extT.inject plus-extT.simps(1) bot-extT-def)
  by (simp-all add: bot-extT-def)
show \(x \leq y \lor y \leq x\)
  by (cases x; cases y) (simp-all add: linear)
show \(-x = (if x = bot then top else bot)\)
  by (cases x) (simp-all add: bot-extT-def top-extT-def)
show \(T :: 'a extT) = top\)
  by (simp add: one-extT-def top-extT-def)
show \(x * y = x \sqcap y\)
  by simp
show \(x^T = x\)
  by simp
show $x^\star = \text{top}$
  by (simp add: top-extT-def)
qed

end

An example of a linearly ordered commutative monoid with a least element is the set of real numbers extended by minus infinity with maximum as aggregation.

datatype real-max-bot =
  MInfty
| R real

instantiation real-max-bot :: linordered-comm-monoid-add-bot
begin

definition bot-real-max-bot ≡ MInfty

fun less-eq-real-max-bot where
  less-eq-real-max-bot MInfty - = True
| less-eq-real-max-bot (R -) MInfty = False
| less-eq-real-max-bot (R x) (R y) = ($x \leq y$)

fun less-real-max-bot where
  less-real-max-bot - MInfty = False
| less-real-max-bot MInfty (R -) = True
| less-real-max-bot (R x) (R y) = ($x < y$)

fun plus-real-max-bot where
  plus-real-max-bot MInfty y = y
| plus-real-max-bot x MInfty = x
| plus-real-max-bot (R x) (R y) = R (max x y)

instance

proof
  fix x y z :: real-max-bot
  show $(x + y) + z = x + (y + z)$
    by (cases x; cases y; cases z) simp-all
  show $x + y = y + x$
    by (cases x; cases y) simp-all
  show $(x < y) = (x \leq y \land \neg y \leq x)$
    by (cases x; cases y) auto
  show $x \leq x$
    by (cases x) simp-all
  show $x \leq y \implies y \leq z \implies x \leq z$
    by (cases x; cases y; cases z) simp-all
  show $x \leq y \implies y \leq x \implies x = y$
    by (cases x; cases y) simp-all
  show $x \leq y \implies z + x \leq z + y$

54
by (cases x; cases y; cases z) simp-all
show \( x \leq y \lor y \leq x \)
  by (cases x; cases y) auto
show \( \text{bot} \leq x \)
  by (cases x) (simp-all add: bot-real-max-bot-def)
show \( \text{bot} + x = x \)
  by (cases x) (simp-all add: bot-real-max-bot-def)
qed

end

5.4 Linearly Ordered Commutative Monoids with a Greatest Element

If a linearly ordered commutative monoid already contains a greatest element which is a unit of aggregation, only a new least element has to be added to obtain a linear aggregation lattice.

class linordered-comm-monoid-add-top = linordered-ab-semigroup-add + order-top +
  assumes top-zero [simp]: top + x = x
begin

sublocale linordered-comm-monoid-add where zero = top
  apply unfold-locales
  by simp

lemma add-decreasing: \( x + y \leq x \)
  using add-left-mono top.extremum by fastforce

lemma t-min: \( x + y \leq \text{min} x y \)
  using add-commute add-decreasing by force

end

datatype 'a extB =
  Bot
| Val 'a

instantiation extB :: (linordered-comm-monoid-add-top)
  linear-aggregation-kleene-algebra
begin

fun plus-extB :: 'a extB ⇒ 'a extB ⇒ 'a extB where
  plus-extB Bot Bot = Val top
| plus-extB Bot (Val x) = Val x
| plus-extB (Val x) Bot = Val x
| plus-extB (Val x) (Val y) = Val (x + y)

fun sup-extB :: 'a extB ⇒ 'a extB ⇒ 'a extB where
sup-extB Bot \ x = \ x \\
| sup-extB (Val \ x) Bot = Val \ x \\
| sup-extB (Val \ x) (Val \ y) = Val (\max \ x \ y)

fun inf-extB :: '\ a extB ⇒ '\ a extB ⇒ '\ a extB where
inf-extB Bot \ = \ Bot \\
| inf-extB (Val \ -) Bot = Bot \\
| inf-extB (Val \ x) (Val \ y) = Val (\min \ x \ y)

fun times-extB :: '\ a extB ⇒ '\ a extB ⇒ '\ a extB where
times-extB \ x \ y = \ x ⊓ \ y

fun uminus-extB :: '\ a extB ⇒ '\ a extB where
uminus-extB Bot = Val top \\
| uminus-extB (Val \ -) = Bot 

fun star-extB :: '\ a extB ⇒ '\ a extB where
star-extB \ = \ Val top

fun conv-extB :: '\ a extB ⇒ '\ a extB where
conv-extB \ x = \ x

definition bot-extB :: '\ a extB where
bot-extB ≡ Bot

definition one-extB :: '\ a extB where
one-extB ≡ Val top

definition top-extB :: '\ a extB where
top-extB ≡ Val top

fun less-eq-extB :: '\ a extB ⇒ '\ a extB ⇒ bool where
less-eq-extB Bot \ = \ True \\
| less-eq-extB (Val \ -) Bot = False \\
| less-eq-extB (Val \ x) (Val \ y) = (\ x \leq \ y)

fun less-extB :: '\ a extB ⇒ '\ a extB ⇒ bool where
less-extB \ x \ y = (\ x \leq \ y \land \neg \ y \leq \ x)

instance
proof
fix \ x \ y \ z :: '\ a extB
show (\ x + \ y) + \ z = \ x + (\ y + \ z) 
  by (cases \ x; cases \ y; cases \ z) (simp-all add: add.assoc)
show \ x + \ y = \ y + \ x 
  by (cases \ x; cases \ y) (simp-all add: add.commute)
show (\ x < \ y) = (\ x \leq \ y \land \neg \ y \leq \ x) 
  by simp
show \ x \leq \ x 
  by (cases \ x) simp-all
show \ x \leq \ y \implies \ y \leq \ z \implies \ x \leq \ z 
  by (cases \ x; cases \ y; cases \ z) simp-all
show \ x \leq \ y \implies \ y \leq \ x \implies \ x = \ y 
  by (cases \ x; cases \ y) simp-all
show \ x \cap \ y \leq \ x 
  by (cases \ x; cases \ y) simp-all
show \ x \cap \ y \leq \ y 

by (cases x; cases y) simp-all
show \( x \leq y \implies x \leq z \implies x \leq y \cap z \)
  by (cases x; cases y; cases z) simp-all
show \( x \leq x \sqcup y \)
  by (cases x; cases y) simp-all
show \( y \leq x \sqcup y \)
  by (cases x; cases y) simp-all
show \( y \leq x \implies z \leq x \implies y \sqcup z \leq x \)
  by (cases x; cases y; cases z) simp-all
show \( \text{bot} \leq x \)
  by (simp add: bot-extB-def)
show \( 1 : x \leq \text{top} \)
  by (cases x) (simp-all add: top-extB-def)
show \( x \neq \text{bot} \land x + \text{bot} \leq y + \text{bot} \implies x + z \leq y + z \)
  apply (cases x; cases y; cases z)
  prefer 6 using 1 apply (metis (mono-tags, lifting) plus-extB.simps(2,4)
top-extB-def add-right-mono less-eq-extB.simps(\_\_) top-zero)
  by (simp-all add: bot-extB-def add-right-mono)
show \( x + y + \text{bot} = x + y \)
  by (cases x; cases y) (simp-all add: bot-extB-def)
show \( x + y = \text{bot} \implies x = \text{bot} \)
  by (cases x; cases y) (simp-all add: bot-extB-def)
show \( x \leq y \lor y \leq x \)
  by (cases x; cases y) (simp-all add: linear)
show \( -x = (\text{if} x = \text{bot} \text{ then} \text{top} \text{ else} \text{bot}) \)
  by (cases x) (simp-all add: bot-extB-def top-extB-def)
show \( 1 :: 'a \text{ extB} = \text{top} \)
  by (simp add: one-extB-def top-extB-def)
show \( x * y = x \cap y \)
  by simp
show \( x^+ = x \)
  by simp
show \( x^+ = \text{top} \)
  by (simp add: top-extB-def)
qed

end

An example of a linearly ordered commutative monoid with a greatest
element is the set of real numbers extended by infinity with minimum as
aggregation.

datatype real-min-top =
  R real
| PInfty

instantiation real-min-top :: linordered-comm-monoid-add-top
begin
definition top-real-min-top ≡ PInfty
fun less-eq-real-min-top where
  less-eq-real-min-top - PInfty = True
  less-eq-real-min-top PInfty (R -) = False
  less-eq-real-min-top (R x) (R y) = (x ≤ y)

fun less-real-min-top where
  less-real-min-top PInfty - = False
  less-real-min-top (R -) PInfty = True
  less-real-min-top (R x) (R y) = (x < y)

fun plus-real-min-top where
  plus-real-min-top PInfty y = y
  plus-real-min-top x PInfty = x
  plus-real-min-top (R x) (R y) = R (min x y)

instance proof
  fix x y z :: real-min-top
  show (x + y) + z = x + (y + z)
    by (cases x; cases y; cases z) simp-all
  show x + y = y + x
    by (cases x; cases y) simp-all
  show (x < y) = (x ≤ y ∧ ¬ y ≤ x)
    by (cases x; cases y) auto
  show x ≤ x
    by (cases x) simp-all
  show x ≤ y ⇒ y ≤ z ⇒ x ≤ z
    by (cases x; cases y; cases z) simp-all
  show x ≤ y ⇒ y ≤ x ⇒ x = y
    by (cases x; cases y) simp-all
  show x ≤ y ⇒ z + x ≤ z + y
    by (cases x; cases y; cases z) simp-all
  show x ≤ y ∨ y ≤ x
    by (cases x; cases y) auto
  show x ≤ top
    by (cases x) (simp-all add: top-real-min-top-def)
  show top + x = x
    by (cases x) (simp-all add: top-real-min-top-def)
qed end

Another example of a linearly ordered commutative monoid with a greatest element is the unit interval of real numbers with any triangular norm (t-norm) as aggregation. Ideally, we would like to show that the unit interval is an instance of linordered-comm-monoid-add-top. However, this class has an addition operation, so the instantiation would require dependent types. We therefore show only the order property in general and a particular in-

58
stance of the class.

typedef (overloaded) unit = {0..1} :: real set by auto

setup-lifting type-definition-unit

instantiation unit :: bounded-linorder begin

lift-definition bot-unit :: unit is 0 by simp

lift-definition top-unit :: unit is 1 by simp

lift-definition less-eq-unit :: unit ⇒ unit ⇒ bool is less-eq .

lift-definition less-unit :: unit ⇒ unit ⇒ bool is less .

instance
  apply intro-classes

end

We give the Lukasiewicz t-norm as a particular instance.

instantiation unit :: linordered-comm-monoid-add-top begin

abbreviation tl :: real ⇒ real ⇒ real where
  tl x y ≡ max (x + y - 1) 0

lemma tl-assoc:
  x ∈ {0..1} ⇒ z ∈ {0..1} ⇒ tl (tl x y) z = tl x (tl y z)
  by auto

lemma tl-top-zero:
  x ∈ {0..1} ⇒ tl 1 x = x
  by auto

lift-definition plus-unit :: unit ⇒ unit ⇒ unit is tl
  by simp

instance
  apply intro-classes
  apply (metis (mono-tags, lifting) plus-unit.rep-eq unit.Rep-unit-inject unit.Rep-unit tl-assoc)
  using unit.Rep-unit-inject plus-unit.rep-eq apply fastforce
5.5 Linearly Ordered Commutative Monoids with a Least Element and a Greatest Element

If a linearly ordered commutative monoid already contains a least element which is a unit of aggregation and a greatest element, it forms a linear aggregation lattice.

class linordered-bounded-comm-monoid-add-bot = 
linordered-comm-monoid-add-bot + order-top
begin
subclass bounded-linorder ..
subclass aggregation-order
  apply unfold-locales
  apply (simp add: add-right-mono)
  apply simp
  by (metis add-0-right add-left-mono bot.extremum bot.extremum-unique)

sublocale linear-aggregation-kleene-algebra where sup = max and inf = min
  and times = min and cone = id and one = top and star = \lambda x . top and
  uminus = \lambda x . if x = bot then top else bot
  apply unfold-locales
  by simp-all

lemma t-top: x + top = top
  by (metis add-right-mono bot.extremum bot-zero top-unique)

lemma add-increasing: x \leq x + y
  using add-left-mono bot.extremum by fastforce

lemma t-max: max x y \leq x + y
  using add-commute add-increasing by force
end

An example of a linearly ordered commutative monoid with a least and a greatest element is the unit interval of real numbers with any triangular conorm (t-conorm) as aggregation. For the reason outlined above, we show just a particular instance of linordered-bounded-comm-monoid-add-bot. Because the plus functions in the two instances given for the unit type are different, we work on a copy of the unit type.

typedef (overloaded) unit2 = {0..1} :: real set
by auto

setup-lifting type-definition-unit2

instantiation unit2 :: bounded-linorder begin

lift-definition bot-unit2 :: unit2 is 0 by simp

lift-definition top-unit2 :: unit2 is 1 by simp

lift-definition less-eq-unit2 :: unit2 ⇒ unit2 ⇒ bool is less-eq.

lift-definition less-unit2 :: unit2 ⇒ unit2 ⇒ bool is less.

instance
apply intro-classes
using bot-unit2.rep-eq top-unit2.rep-eq less-eq-unit2.rep-eq less-unit2.rep-eq unit2.Rep-unit2-inject unit2.Rep-unit2 by auto

end

We give the product t-conorm as a particular instance.

instantiation unit2 :: linordered-bounded-comm-monoid-add-bot begin

abbreviation sp :: real ⇒ real ⇒ real where
sp x y ≡ x + y - x * y

lemma sp-assoc:
sp (sp x y) z = sp x (sp y z) by (unfold left-diff-distrib right-diff-distrib distrib-left distrib-right) simp

lemma sp-mono:
assumes z ∈ {0..1}
and x ≤ y
shows sp z x ≤ sp z y

proof –
have z + (1 - z) * x ≤ z + (1 - z) * y using assms mult-left_mono by fastforce
thus ?thesis
by (unfold left-diff-distrib right-diff-distrib distrib-left distrib-right) simp

qed

lift-definition plus-unit2 :: unit2 ⇒ unit2 ⇒ unit2 is sp

proof –
fix x y :: real

61
assume 1: \( x \in \{0..1\} \)
assume 2: \( y \in \{0..1\} \)
have \( x - x \ast y \leq 1 - y \)
using 1 2 by (metis (full-types) atLeastAtMost-iff diff-ge-0-iff-ge
left-diff-distrib' mult.commute mult.left-neutral mult-left-le)

hence 3: \( x + y - x \ast y \leq 1 \)
by simp
have \( y \ast (x - 1) \leq 0 \)
using 1 2 by (meson atLeastAtMost-iff le-iff-diff-le-0 mult-nonneg-nonpos)
hence \( x + y - x \ast y \geq 0 \)
using 1 by (metis (no-types) atLeastAtMost-iff diff-diff-eq2 diff-ge-0-iff-ge
left-diff-distrib mult.commute mult.left-neutral order-trans)
thus \( x + y - x \ast y \in \{0..1\} \)
using 3 by simp

qed

instance
apply intro-classes
apply (metis (mono-tags, lifting) plus-unit2.rep-eq unit2.Rep-unit2-inject
sp-assoc)
using unit2.Rep-unit2-inject plus-unit2.rep-eq apply fastforce
using sp-mono unit2.Rep-unit2 less-eq-unit2.rep-eq plus-unit2.rep-eq apply simp
using bot-unit2.rep-eq unit2.Rep-unit2-inject plus-unit2.rep-eq by fastforce

end

5.6 Constant Aggregation

Any linear order with a constant element extended by new least and greatest
elements forms a linear aggregation lattice where the aggregation returns the
given constant.

class pointed-linorder = linorder +
fixes const :: 'a

datatype 'a extC =
  Bot
| Val 'a
| Top

instantiation extC :: (pointed-linorder) linear-aggregation-kleene-algebra
begin

fun plus-extC :: 'a extC ⇒ 'a extC ⇒ 'a extC where
  plus-extC x y = Val const

fun sup-extC :: 'a extC ⇒ 'a extC ⇒ 'a extC where
  sup-extC Bot x = x
| sup-extC (Val x) Bot = Val x
| sup-extC (Val x) (Val y) = Val (max x y)
sup-extC (Val -) Top = Top
sup-extC Top - = Top

fun inf-extC :: 'a extC ⇒ 'a extC ⇒ 'a extC where
inf-extC Bot - = Bot
inf-extC (Val -) Bot = Bot
inf-extC (Val x) (Val y) = Val (min x y)
inf-extC (Val x) Top = Val x
inf-extC Top x = x

fun times-extC :: 'a extC ⇒ 'a extC ⇒ 'a extC where times-extC x y = x \odot y

fun uminus-extC :: 'a extC where
uminus-extC Bot = Top
uminus-extC (Val -) = Bot
uminus-extC Top = Bot

fun star-extC :: 'a extC ⇒ 'a extC where star-extC - = Top

fun conv-extC :: 'a extC ⇒ 'a extC where conv-extC x = x

definition bot-extC :: 'a extC where bot-extC ≡ Bot
definition one-extC :: 'a extC where one-extC ≡ Top
definition top-extC :: 'a extC where top-extC ≡ Top

fun less-eq-extC :: 'a extC ⇒ 'a extC ⇒ bool where
less-eq-extC Bot - = True
less-eq-extC (Val -) Bot = False
less-eq-extC (Val x) (Val y) = (x \leq y)
less-eq-extC (Val -) Top = True
less-eq-extC Top Bot = False
less-eq-extC Top (Val -) = False
less-eq-extC Top Top = True

fun less-extC :: 'a extC ⇒ 'a extC ⇒ bool where less-extC x y = (x \leq y \land \neg y \leq x)

instance
proof
fix x y z :: 'a extC
show (x + y) + z = x + (y + z)
  by simp
show x + y = y + x
  by simp
show (x < y) = (x \leq y \land \neg y \leq x)
  by simp
show x \leq x
  by (cases x) simp-all
show x \leq y \implies y \leq z \implies x \leq z

63
by (cases x; cases y; cases z) simp-all
show \( x \leq y \implies y \leq x \implies x = y \)
  by (cases x; cases y) simp-all
show \( x \cap y \leq x \)
  by (cases x; cases y) simp-all
show \( x \cap y \leq y \)
  by (cases x; cases y) simp-all
show \( x \leq y \implies x \leq z \implies y \cap z \)
  by (cases x; cases y; cases z) simp-all
show \( x \leq x \sqcup y \)
  by (cases x; cases y) simp-all
show \( y \leq x \sqcup y \)
  by (cases x; cases y) simp-all
show \( y \leq x \implies z \leq x \implies y \sqcup z \leq x \)
  by (cases x; cases y; cases z) simp-all
show \( \text{bot} \leq x \)
  by (simp add: bot-extC-def)
show \( x \leq \text{top} \)
  by (cases x) (simp-all add: top-extC-def)
show \( x \neq \text{bot} \land x + \text{bot} \leq y + \text{bot} \implies x + z \leq y + z \)
  by simp
show \( x + y + \text{bot} = x + y \)
  by simp
show \( x + y = \text{bot} \implies x = \text{bot} \)
  by (simp add: bot-extC-def)
show \( x \leq y \lor y \leq x \)
  by (cases x; cases y) (simp-all add: linear)
show \( -x = (\text{if } x = \text{bot} \text{ then } \text{top} \text{ else } \text{bot}) \)
  by (cases x) (simp-all add: bot-extC-def top-extC-def)
show \( 1::'a extC) = \text{top} \)
  by (simp add: one-extC-def top-extC-def)
show \( x * y = x \cap y \)
  by simp
show \( x^T = x \)
  by simp
show \( x^* = \text{top} \)
  by (simp add: top-extC-def)

qed

end

An example of a linear order is the set of real numbers. Any real number can be chosen as the constant.

instantiation real :: pointed-linorder
begin

instance ..

end

64
The following instance shows that any linear order with a constant forms a linearly ordered commutative semigroup with the alpha-median operation as aggregation. The alpha-median of two elements is the median of these elements and the given constant.

fun median3 :: 'a::ord ⇒ 'a ⇒ 'a ⇒ 'a where
  median3 x y z =
  (if x ≤ y ∧ y ≤ z then y else
   if x ≤ z ∧ z ≤ y then z else
   if y ≤ x ∧ x ≤ z then x else
   if y ≤ z ∧ z ≤ x then z else
   if z ≤ x ∧ x ≤ y then x else y)

interpretation alpha-median: linordered-ab-semigroup-add where plus = median3 const and less-eq = less-eq and less = less

proof
  fix a b c :: 'a
  show median3 const (median3 const a b) c = median3 const a (median3 const b c)
  by (cases const ≤ a; cases const ≤ b; cases const ≤ c; cases a ≤ b; cases a ≤ c; cases b ≤ c) auto
  show median3 const a b = median3 const b a
  by (cases const ≤ a; cases const ≤ b; cases a ≤ b) auto
  assume a ≤ b
  thus median3 const c a ≤ median3 const c b
  by (cases const ≤ a; cases const ≤ b; cases const ≤ c; cases a ≤ c; cases b ≤ c) auto
qed

5.7 Counting Aggregation

Any linear order extended by new least and greatest elements and a copy of the natural numbers forms a linear aggregation lattice where the aggregation counts non-⊥ elements using the copy of the natural numbers.

datatype 'a extN =
  Bot
  | Val 'a
  | N nat
  | Top

instantiation extN :: (linorder) linear-aggregation-kleene-algebra

begin

fun plus-extN :: 'a extN ⇒ 'a extN ⇒ 'a extN where
  plus-extN Bot Bot = N 0
  | plus-extN Bot (Val -) = N 1
  | plus-extN Bot (N y) = N y
  | plus-extN Bot Top = N 1
  | plus-extN (Val -) Bot = N 1
\[ \text{fun } \text{sup-extN} :: 'a extN \Rightarrow 'a extN \Rightarrow 'a extN \text{ where} \]
\[
\begin{align*}
\text{sup-extN } 
& \text{Bot } x = x \\
& \text{sup-extN } (\text{Val } x) \text{ Bot } = \text{Val } x \\
& \text{sup-extN } (\text{Val } x) (\text{Val } y) = \text{Val } (\text{max } x y) \\
& \text{sup-extN } (\text{Val } x) (N y) = N y \\
& \text{sup-extN } (\text{Val } -) \text{ Top } = \text{Top} \\
& \text{sup-extN } (N x) \text{ Bot } = N x \\
& \text{sup-extN } (N x) (\text{Val } -) = N x \\
& \text{sup-extN } (N x) (N y) = N (\text{max } x y) \\
& \text{sup-extN } (N -) \text{ Top } = \text{Top} \\
& \text{sup-extN } \text{Top } y = y
\end{align*}
\]

\text{fun } \text{inf-extN} :: 'a extN \Rightarrow 'a extN \Rightarrow 'a extN \text{ where}
\[
\begin{align*}
\text{inf-extN } \text{Bot } - = \text{Bot} \\
& \text{inf-extN } (\text{Val } -) \text{ Bot } = \text{Bot} \\
& \text{inf-extN } (\text{Val } x) (\text{Val } y) = \text{Val } (\text{min } x y) \\
& \text{inf-extN } (\text{Val } x) (N -) = \text{Val } x \\
& \text{inf-extN } (\text{Val } x) \text{ Top } = \text{Val } x \\
& \text{inf-extN } (N -) \text{ Bot } = \text{Bot} \\
& \text{inf-extN } (N -) (\text{Val } y) = \text{Val } y \\
& \text{inf-extN } (N x) (N y) = N (\text{min } x y) \\
& \text{inf-extN } (N x) \text{ Top } = N x \\
& \text{inf-extN } \text{Top } y = y
\end{align*}
\]

\text{fun } \text{times-extN} :: 'a extN \Rightarrow 'a extN \Rightarrow 'a extN \text{ where times-extN } x y = x \cap y

\text{fun } \text{uminus-extN} :: 'a extN \Rightarrow 'a extN \text{ where}
\[
\begin{align*}
\text{uminus-extN } \text{Bot } & = \text{Top} \\
& \text{uminus-extN } (\text{Val } -) = \text{Bot} \\
& \text{uminus-extN } (N -) = \text{Bot} \\
& \text{uminus-extN } \text{Top } = \text{Bot}
\end{align*}
\]

\text{fun } \text{star-extN} :: 'a extN \Rightarrow 'a extN \text{ where star-extN } - = \text{Top}

\text{fun } \text{conv-extN} :: 'a extN \Rightarrow 'a extN \text{ where conv-extN } x = x

\text{definition } \text{bot-extN} :: 'a extN \text{ where bot-extN } \equiv \text{Bot}

definition \( \text{one-extN} :: 'a \text{ extN} \ \text{where} \ \text{one-extN} \equiv \text{Top} \)

definition \( \text{top-extN} :: 'a \text{ extN} \ \text{where} \ \text{top-extN} \equiv \text{Top} \)

fun \( \text{less-eq-extN} :: 'a \text{ extN} \Rightarrow 'a \text{ extN} \Rightarrow \text{bool} \ \text{where} \)
\( \text{less-eq-extN} \ \text{Bot} - = \text{True} \)
\( \text{less-eq-extN} \ (\text{Val} \ x) \ (\text{Val} \ y) = (x \leq y) \)
\( \text{less-eq-extN} \ (\text{Val} \ x) \ \text{Top} = \text{True} \)
\( \text{less-eq-extN} \ (\text{Val} \ y) \ \text{Bot} = \text{False} \)
\( \text{less-eq-extN} \ (\text{N} \ x) \ (\text{N} \ y) = (x \leq y) \)
\( \text{less-eq-extN} \ (\text{N} \ x) \ \text{Top} = \text{True} \)
\( \text{less-eq-extN} \ \text{Top} \ (\text{Val} \ y) = \text{False} \)
\( \text{less-eq-extN} \ \text{Top} \ (\text{N} \ y) = \text{False} \)
\( \text{less-eq-extN} \ \text{Top} \ \text{Top} = \text{True} \)

fun \( \text{less-extN} :: 'a \text{ extN} \Rightarrow 'a \text{ extN} \Rightarrow \text{bool} \ \text{where} \)
\( \text{less-extN} \ x \ y = (x \leq y \land \neg y \leq x) \)

instance

proof

fix \( x \ y \ z :: 'a \text{ extN} \)

show \((x + y) + z = x + (y + z)\)
  by (cases \(x\); cases \(y\); cases \(z\)) simp-all

show \(x + y = y + x\)
  by (cases \(x\); cases \(y\)) simp-all

show \((x < y) = (x \leq y \land \neg y \leq x)\)
  by simp

show \(x \leq x\)
  by (cases \(x\)) simp-all

show \(x \leq y \Longrightarrow y \leq z \Longrightarrow x \leq z\)
  by (cases \(x\); cases \(y\); cases \(z\)) simp-all

show \(x \leq y \Longrightarrow x \leq x \Longrightarrow x = y\)
  by (cases \(x\); cases \(y\)) simp-all

show \(x \sqcap y \leq x\)
  by (cases \(x\); cases \(y\)) simp-all

show \(x \sqcap y \leq y\)
  by (cases \(x\); cases \(y\)) simp-all

show \(x \sqcap y \leq z \Longrightarrow x \leq z \Longrightarrow y \leq z \Longrightarrow x \sqcap z \leq x\)
  by (cases \(x\); cases \(y\); cases \(z\)) simp-all

show \(y \leq x \sqcup y\)
  by (cases \(x\); cases \(y\)) simp-all

show \(y \leq x \Longrightarrow z \leq x \Longrightarrow y \sqcup z \leq x\)
  by (cases \(x\); cases \(y\); cases \(z\)) simp-all

show \(\text{bot} \leq x\)

67
by (simp add: bot-extN-def)
show \( x \leq \top \)
  by (cases \( x \)) (simp-all add: top-extN-def)
show \( x \neq \bot \wedge x + \bot \leq y + \bot \longrightarrow x + z \leq y + z \)
  by (cases \( x \); cases \( y \); cases \( z \)) (simp-all add: bot-extN-def)
show \( x + y + \bot = x + y \)
  by (cases \( x \); cases \( y \)) (simp-all add: bot-extN-def)
show \( x + y = \bot \longrightarrow x = \bot \)
  by (cases \( x \); cases \( y \)) (simp-all add: bot-extN-def)
show \( x \leq y \lor y \leq x \)
  by (cases \( x \); cases \( y \)) (simp-all add: linear)
show \(-x = (if x = \bot then \top else \bot)\)
  by (cases \( x \)) (simp-all add: bot-extN-def top-extN-def)
show \((1::'a extN) = \top\)
  by (simp add: one-extN-def top-extN-def)
show \( x * y = x \cap y \)
  by simp
show \( x^T = x \)
  by simp
show \( x^* = \top \)
  by (simp add: top-extN-def)
qed

end

end

6 Hoare Logic for Total Correctness

theory Hoare-Logic
imports Main
begin

This theory is based on Isabelle/HOL’s Hoare/Hoare-Logic.thy written
by L. P. Nieto and T. Nipkow. We have extended it to total-correctness
proofs. We added corresponding modifications to hoare-syntax.ML and
hoare-tac.ML.

type-synonym ‘a bexp = ‘a set
type-synonym ‘a assn = ‘a set
type-synonym ‘a var = ‘a \Rightarrow \text{nat}

datatype ‘a com =
  Basic 'a \Rightarrow 'a
  | Seq 'a com 'a com
  | Cond 'a bexp 'a com 'a com
  | While 'a bexp 'a assn 'a var 'a com

68
syntax (xsymbols)
-whilePC :: 'a bexp ⇒ 'a assn ⇒ 'a com ⇒ 'a com ((1WHILE / INV {¬})
//DO / OD) [0,0,0] 61)

translations
WHILE b INV {x} DO c OD => WHILE b INV {x} VAR {0} DO c OD

abbreviation annskip (SKIP) where SKIP == Basic id

type-synonym 'a sem = 'a => 'a sem => bool

inductive Sem :: 'a com ⇒ 'a sem
where
Sem (Basic f) s (f s)
| Sem c1 s s'' => Sem c2 s'' s' => Sem (c1;c2) s s'
| s ∈ b => Sem c1 s s' => Sem (IF b THEN c1 ELSE c2 FI) s s'
| s ∉ b => Sem c2 s s' => Sem (IF b THEN c1 ELSE c2 FI) s s'
| s ∈ b => Sem (While b x y c) s s'
| s ∉ b => Sem (While b x y c) s s'' => Sem (While b x y c) s'' s'

inductive-cases [elim!]:
Sem (Basic f) s s' Sem (c1;c2) s s'
Sem (IF b THEN c1 ELSE c2 FI) s s'

lemma Sem-deterministic:
assumes Sem c s s1
and Sem c s s2
shows s1 = s2
proof –
have Sem c s s1 => (∀ s2. Sem c s s2 => s1 = s2)
  by (induct rule: Sem.induct) (subst Sem.simps, blast)+
thus ?thesis
using assms by simp
qed

definition Valid :: 'a bexp ⇒ 'a com ⇒ 'a bexp ⇒ bool
where Valid p c q <= ((∀ s' . Sem c s s' => s ∈ p => s' ∈ q)
definition ValidTC :: 'a bexp ⇒ 'a com ⇒ 'a bexp ⇒ bool
where ValidTC p c q ≡ ∀ s . s ∈ p => (∃ t . Sem c s t ∧ t ∈ q)

lemma tc-implies-pc:
ValidTC p c q => Valid p c q
by (metis Sem-deterministic Valid-def ValidTC-def)

lemma tc-extract-function:
ValidTC p c q => ∃ f . ∀ s . s ∈ p => f s ∈ q
by (metis ValidTC-def)
lemma SkipRule: \( p \subseteq q \Rightarrow \text{Valid} \ (\text{Basic id}) \ q \) by (auto simp:Valid-def)

lemma BasicRule: \( p \subseteq \{ s. \ f \ s \in q \} \Rightarrow \text{Valid} \ (\text{Basic f}) \ q \) by (auto simp:Valid-def)

lemma SeqRule: \( \text{Valid} \ P \ c1 \ Q \Rightarrow \text{Valid} \ Q \ c2 \ R \Rightarrow \text{Valid} \ P \ (c1;c2) \ R \) by (auto simp:Valid-def)

lemma CondRule:
\[
p \subseteq \{ s. \ (s \in b \rightarrow s \in w) \land (s \notin b \rightarrow s \in w')\} \\ightarrow \text{Valid} \ w \ c1 \ q \Rightarrow \text{Valid} \ w' \ c2 \ q \Rightarrow \text{Valid} \ p \ (\text{Cond} \ b \ c1 \ c2) \ q
\] by (auto simp:Valid-def)

lemma While-aux:
\[
\text{assumes} \ Sem \ (\text{WHILE} \ b \ \text{INV} \ \{i\} \ \text{VAR} \ \{v\} \ DO \ c \ OD) \ s \ s' \\
\text{shows} \ \forall s \ s'. \ Sem \ c \ s \ s' \Rightarrow s \in I \land s \in b \rightarrow s' \in I \Rightarrow \\
\ s \in I \Rightarrow s' \in I \land s' \notin b
\] using assms

70
by (induct WHILE b INV \{i\} VAR \{v\} DO OD s s') auto

lemma WhileRule:
\[
p \subseteq i \implies \text{Valid } (i \cap b) \implies i \cap (-b) \subseteq q \implies \text{Valid } p \ (\text{While } b \ i \ v \ c) \ q
\]
apply (clarsimp simp: Valid-def)
apply (drule While-aux)
apply assumption
apply blast
apply blast
done

lemma SkipRuleTC:
assumes \[
p \subseteq q \] shows \[
\text{ValidTC } p \ (\text{Basic } id) \ q
\]
by (metis assms Sem.intros(1) ValidTC-def id-apply set-mp)

lemma BasicRuleTC:
assumes \[
p \subseteq \{s. f s \in q\}
\]
shows \[
\text{ValidTC } p \ (\text{Basic } f) \ q
\]
by (metis assms Ball-Collect Sem.intros(1) ValidTC-def)

lemma SeqRuleTC:
assumes \[
\text{ValidTC } p \ c1 q \quad \text{and} \quad \text{ValidTC } q \ c2 r
\]
shows \[
\text{ValidTC } p \ (c1; c2) \ r
\]
by (meson assms Sem.intros(2) ValidTC-def)

lemma CondRuleTC:
assumes \[
p \subseteq \{s. (s \in b \implies s \in w) \land (s \notin b \implies s \in w')\}
\]
and \[
\text{ValidTC } w \ c1 q \quad \text{and} \quad \text{ValidTC } w' \ c2 q
\]
shows \[
\text{ValidTC } p \ (\text{Cond } b \ c1 \ c2) \ q
\]
proof (unfold ValidTC-def, rule allI)
fix s
show \[
s \in p \implies (\exists t. \text{Sem } (\text{Cond } b \ c1 \ c2) s t \land t \in q)
\]
apply (cases s \in b)
apply (metis (mono-tags, lifting) assms(1,2) Ball-Collect Sem.intros(3) ValidTC-def)
by (metis (mono-tags, lifting) assms(1,3) Ball-Collect Sem.intros(4) ValidTC-def)
qed

lemma WhileRuleTC:
assumes \[
p \subseteq i
\]
and \[
\forall n::\text{nat} . \ \text{ValidTC } (i \cap b \cap \{s. \ v \ s = n\}) \ c (i \cap \{s. \ v \ s < n\})
\]
and \[
i \cap \text{uminus } b \subseteq q
\]
shows \[
\text{ValidTC } p \ (\text{While } b \ i \ v \ c) \ q
\]
proof 


fix s n
have s ∈ i ∧ v s = n → (∃ t . Sem (While b i v c) s t ∧ t ∈ q)
proof (induction n arbitrary; s rule: less-induct)
  fix n :: nat
  fix s :: 'a
  assume 1: (∀(m::nat) s::'a . m < n → s ∈ i ∧ v s = m → (∃ t . Sem (While b i v c) s t ∧ t ∈ q))
  show s ∈ i ∧ v s = n → (∃ t . Sem (While b i v c) s t ∧ t ∈ q)
  proof (rule impI, cases s ∈ b)
    assume 2: s ∈ b and s ∈ i ∧ v s = n
    hence s ∈ i ∩ b ∩ \{ s . v s = n\}
    using assms(1) by auto
    hence ∃ t . Sem c s t ∧ t ∈ i ∩ \{ s . v s < n\}
    by (metis assms(2) ValidTC-def)
    from this obtain t where 3: Sem c s t ∧ t ∈ i ∩ \{ s . v s < n\}
    by auto
    hence ∃ u . Sem (While b i v c) t u ∧ u ∈ q
    using 1 by auto
    thus ∃ t . Sem (While b i v c) s t ∧ t ∈ q
    using 2 3 Sem.intros(6) by force
next
  assume s ∉ b and s ∈ i ∧ v s = n
  thus ∃ t . Sem (While b i v c) s t ∧ t ∈ q
  using Sem.intros(5) assms(3) by fastforce
qed
qed

thus \#thesis
  using assms(1) ValidTC-def by force
qed

lemma Compl-Collect: − (Collect b) = \{ x . ∼(b x)\}
by blast

lemmas AbortRule = SkipRule — dummy version
lemmas AbortRuleTC = SkipRuleTC — dummy version
ML-file hoare-tac.ML

method-setup vcg = ⟨
  Scan.succeed (fn ctxt => SIMPLE-METHOD' (Hoare.hoare-tac ctxt (K all-tac)))
⟩
verification condition generator

method-setup vcg-simp = ⟨
  Scan.succeed (fn ctxt =>
  SIMPLE-METHOD' (Hoare.hoare-tac ctxt (asm-full-simp-tac ctxt)))
⟩
verification condition generator plus simplification

method-setup vcg-tc = ⟨
7 Examples using Hoare Logic for Total Correctness

theory Hoare-Logic-Examples

imports Hoare-Logic

begin

This theory demonstrates a few simple partial- and total-correctness proofs. The first example is taken from HOL/Hoare/Examples.thy written by N. Galm. We have added the invariant $m \leq a$.

lemma multiply-by-add: VARS $m \ s \ a \ b$
{ $a=A \land b=B$ }
$m := 0; \ s := 0;$
WHILE $m \neq a$
INV { $s=m\cdot b \land a=A \land b=B \land m \leq a$ }
DO $s := s+b; \ m := m+(1::nat) \ OD$
{ $s = A\cdot B$ }
by vcg-simp

Here is the total-correctness proof for the same program. It needs the additional invariant $m \leq a$.

lemma multiply-by-add-tc: VARS $m \ s \ a \ b$
{ $a=A \land b=B$ }
$m := 0; \ s := 0;$
WHILE $m \neq a$
INV { $s=m\cdot b \land a=A \land b=B \land m \leq a$ }
VAR { $a-m$ }
DO $s := s+b; \ m := m+(1::nat) \ OD$
{ $s = A\cdot B$ }
apply vcg-tc-simp
by auto

Next, we prove partial correctness of a program that computes powers.

lemma power: VARS $(x::nat) \ n \ p \ i$
{ $0 \leq n$ }

Here is its total-correctness proof.

**lemma** power-tc: \(\text{VARS } (x::\text{nat}) \ n \ p \ i\)

\[
p := 1; \quad i := 0; \quad \text{WHILE } i < n \\
\text{INV } \{ p = x^i \land i \leq n \} \\
\text{DO } p := p \ast x; \\
\quad i := i + 1 \\
\text{OD} \\
\{ p = x^n \}
\]

**apply** vcg-simp
**by** auto

8 Minimum Spanning Tree Algorithms

In this theory we prove the total-correctness of Kruskal’s and Prim’s minimum spanning tree algorithms. Specifications and algorithms work in Stone-Kleene relation algebras extended by operations for aggregation and minimisation. The algorithms are implemented in a simple imperative language and their proof uses Hoare Logic. The correctness proofs are discussed in [1, 4, 5].
8.1 Kruskal’s Minimum Spanning Tree Algorithm

The total-correctness proof of Kruskal’s minimum spanning tree algorithm uses the following steps [5]. We first establish that the algorithm terminates and constructs a spanning tree. This is a constructive proof of the existence of a spanning tree; any spanning tree algorithm could be used for this. We then conclude that a minimum spanning tree exists. This is necessary to establish the invariant for the actual correctness proof, which shows that Kruskal’s algorithm produces a minimum spanning tree.

**definition** spanning-forest \(fg\) \(≡\) forest \(f\) \(∧\) components \(g\) \(≤\) forest-components \(f\) \(∧\) regular \(f\)

**definition** minimum-spanning-forest \(fg\) \(≡\) spanning-forest \(fg\) \(∧\) \((∀u . \text{spanning-forest } ug \rightarrow \text{sum } (f \cap g) \leq \text{sum } (u \cap g))\)

**definition** kruskal-spanning-invariant \(fg\) \(h\) \(≡\) symmetric \(g\) \(∧\) \(h\) \(=\) \(h\) \(\land\) spanning-forest \(f\) \((-h \cap g)\)

**definition** kruskal-invariant \(fg\) \(h\) \(≡\) kruskal-spanning-invariant \(fg\) \(h\) \(\land\) \((∃w . \text{minimum-spanning-forest } wg \land f \leq w \sqcup w^T)\)

We first show two verification conditions which are used in both correctness proofs.

**lemma** kruskal-vc-1:
assumes symmetric \(g\)
shows kruskal-spanning-invariant bot \(g\) \(g\)
proof (unfold kruskal-spanning-invariant-def, intro conjI)
show symmetric \(g\) using assms by simp
next
show \(g = g^T\) using assms by simp
next
show \(g \cap + g = g\) using inf.sup-monoid.add-commute selection-closed-id by simp
next
show spanning-forest bot \((-g \cap g)\)
using star.circ-transitive-equal spanning-forest-def by simp
qed

**lemma** kruskal-vc-2:
assumes kruskal-spanning-invariant \(fg\) \(h\)
and \(h \neq \text{bot}\)
and card \{ \(x . \) regular \(x \land x \leq -h\} = n\)
shows \((\text{minarc } h \leq -\text{forest-components } f \rightarrow \text{kruskal-spanning-invariant } (((f \sqcap -(\text{top } \ast \text{minarc } h \ast f^T)) \sqcup (f \sqcap \text{top } \ast \text{minarc } h \ast f^T)^T \sqcup \text{minarc } h) g (h \sqcap -\text{minarc } h \sqcap -\text{minarc } h^T) \land \text{card } \{ \(x . \) regular \(x \land x \leq -h \land x \leq -\text{minarc } h \land x \leq -\text{minarc } h^T \} < n\} \land \neg \text{minarc } h \leq -\text{forest-components } f \rightarrow \text{kruskal-spanning-invariant } fg (h \sqcap -\text{minarc } h \sqcap -\text{minarc } h^T)\)
\[ \land \text{card}\{ x . \text{regular } x \land x \leq \iota - h \land x \leq \iota - \minarc h \} < n \]

**proof**

let \(\varepsilon = \minarc h\)

let \(\mathcal{F} = (f \cap \iota - (\text{top} \ast \varepsilon \ast f^\ast)) \cup (f \cap \text{top} \ast \varepsilon \ast f^T) \cup \varepsilon\)

let \(\mathcal{H} = h \cap \iota - e \cap \iota - \varepsilon^T\)

let \(\mathcal{F} = \text{forest-components } f\)

let \(\mathcal{N}_1 = \text{card}\{ x . \text{regular } x \land x \leq \iota - h \}\)

let \(\mathcal{N}_2 = \text{card}\{ x . \text{regular } x \land x \leq \iota - h \land x \leq \iota - e \land x \leq \iota - \varepsilon^T \}\)

**have** 1: regular \(f \land \text{regular } \varepsilon\)

by (metis assms(1) kruskal-spanning-invariant-def spanning-forest-def minarc-regular)

**hence** 2: regular \(\mathcal{F} \land \text{regular } \varepsilon^T\)

using regular-closed-star regular-conv-closed regular-mult-closed by simp

**have** 3: \(\neg \varepsilon \leq -\varepsilon\)

using assms(2) inf.orderE minarc-bot-iff by fastforce

**have** \(\mathcal{N}_2 < \mathcal{N}_1\)

apply (rule psubset-card-mono)

using finite-regular apply simp

using 1 3 kruskal-spanning-invariant-def minarc-below by auto

**hence** 4: \(\mathcal{N}_2 < n\)

using assms(3) by simp

**show** \((\varepsilon \leq -\varepsilon^T \longrightarrow \text{kruskal-spanning-invariant } \mathcal{F} \land \mathcal{N}_2 < n) \land (\neg \varepsilon \leq -\varepsilon^T \longrightarrow \text{kruskal-spanning-invariant } \mathcal{F} \land \mathcal{N}_2 < n)\)

**proof** (rule conjI)

**have** 5: injective \(\mathcal{F}\)

using assms(1) kruskal-spanning-invariant-def spanning-forest-def apply simp

apply (simp add: covector-mult-closed)

apply (simp add: comp-associative comp-isotone star.right-plus-below-circ)

apply (meson mult-left-isotone order-lesseq-imp star-outer-increasing top.extremum)

using assms(1,2) kruskal-spanning-invariant-def kruskal-injective-inv-2

minarc-arc spanning-forest-def apply simp

using assms(2) arc-injective minarc-arc apply blast

using assms(1,2) kruskal-spanning-invariant-def kruskal-injective-inv-3

minarc-arc spanning-forest-def by simp

**show** \(\varepsilon \leq -\varepsilon^T \longrightarrow \text{kruskal-spanning-invariant } \mathcal{F} \land \mathcal{N}_2 < n\)

**proof**

assume 6: \(\varepsilon \leq -\varepsilon^T\)

**have** 7: equivalence \(\mathcal{F}\)

using assms(1) kruskal-spanning-invariant-def

forest-components-equivalence spanning-forest-def by simp

**have** \(\varepsilon^T \ast \text{top} \ast \varepsilon^T = \varepsilon^T\)

using assms(2) by (simp add: arc-top-arc minarc-arc)

**hence** \(\varepsilon^T \ast \text{top} \ast \varepsilon^T \leq -\varepsilon^T\)

using 6 7 conv-complement conv-isotone by fastforce

**hence** 8: \(\varepsilon \ast \varepsilon^T \ast \varepsilon = \bot\)
using le-bot triple-schroeder-p by simp

show kruskal-spanning-invariant ?f g ?h ∧ ?n2 < n

proof (unfold kruskal-spanning-invariant-def, intro conjI)
  show symmetric g
    using assms(1) kruskal-spanning-invariant-def by simp

next
  show ?h = ?h^T
    using assms(1) by (simp add: conv-complement conv-dist-inf
    inf-commute inf-left-commute kruskal-spanning-invariant-def)

next
  show spanning-forest ?f (¬?h ⊓ g)
    proof (unfold spanning-forest-def, intro conjI)
      show injective ?f
        using 5 by simp

next
  show acyclic ?f
    apply (rule kruskal-acyclic-inv)
    using assms(1) kruskal-spanning-invariant-def spanning-forest-def
    apply simp
    apply (simp add: covector-mult-closed)
    using 8 assms(1) kruskal-spanning-invariant-def spanning-forest-def
    kruskal-acyclic-inv-1 apply simp
    using 8 apply (metis comp-associative mult-left-sub-dist-sup-left
    star.circ-loop-fixpoint sup-commute le-bot)
    using 6 by (simp add: p-antitone-iff)

next
  show ?f ≤ ¬(¬?h ⊓ g)
    apply (rule kruskal-subgraph-inv)
    using assms(1) kruskal-spanning-invariant-def spanning-forest-def
    apply simp
    using assms(1) kruskal-spanning-invariant-def apply simp

next
  show components (¬?h ⊓ g) ≤ forest-components ?f
    apply (rule kruskal-spanning-invariant-inv)
    using 5 apply simp
    using 1 regular-closed-star regular-cone-closed regular-mult-closed
    apply simp
    using 1 apply simp
    using assms(1) kruskal-spanning-invariant-def spanning-forest-def by simp

next
  show regular ?f
using 2 by simp

qed

next

show ?n2 < n

using 4 by simp

qed

qed

next

show \neg ?e \leq - ?F \rightarrow \text{kruskal-spanning-invariant } f \ g \ ?h \ \land \ ?n2 < n

proof

assume \neg ?e \leq - ?F

hence 9: ?e \leq ?F

using 2 assms(2) arc-in-partition minarc-arc by fastforce

show \text{kruskal-spanning-invariant } f \ g \ ?h \ \land \ ?n2 < n

proof (unfold \text{kruskal-spanning-invariant-def}, intro conj1)

show symmetric g

using assms(1) \text{kruskal-spanning-invariant-def} by simp

next

show \ ?h = \ ?h^T

using assms(1) by (simp add: conv-complement conv-dist-inf

inf-commute inf-left-commute kruskal-spanning-invariant-def)

next

show g \cap \neg \neg ?h = ?h

using 1 2 by (metis (hide-lams) assms(1) \text{kruskal-spanning-invariant-def}

inf-assoc pp-dist-inf)

next

show \text{spanning-forest } f \ (\neg ?h \cap g)

proof (unfold \text{spanning-forest-def}, intro conj1)

show injective f

using assms(1) \text{kruskal-spanning-invariant-def} \text{spanning-forest-def} by simp

next

show acyclic f

using assms(1) \text{kruskal-spanning-invariant-def} \text{spanning-forest-def} by simp

next

have f \leq \neg (\neg (\neg \neg h \cap \?g))

using assms(1) \text{kruskal-spanning-invariant-def} \text{spanning-forest-def} by simp

also have ... \leq \neg (\neg \neg h \cap \?g)

using comp-inf mult-right-isotone inf sup-monoid add-commute

inf-left-commute p-antitone-inf pp-isotone by presburger

finally show f \leq \neg (\neg \neg h \cap \?g)

by simp

next

show \text{components } (\neg \neg h \cap \?g) \leq \?F

apply (rule \text{kruskal-spanning-ineq-1})

using 9 apply simp

using 1 apply simp

78
using assms(1) kruskal-spanning-invariant-def spanning-forest-def
apply simp
using assms(1) kruskal-spanning-invariant-def
forest-components-equivalence spanning-forest-def by simp
next
  show regular f
  using 1 by simp
qed
next
  show ?n2 < n
  using 4 by simp
qed
qed
qed

The following result shows that Kruskal’s algorithm terminates and constructs a spanning tree. We cannot yet show that this is a minimum spanning tree.

theorem kruskal-spanning:
  VARS e f h
  [ symmetric g ]
  f := bot;
  h := g;
  WHILE h ≠ bot
    INV { kruskal-spanning-invariant f g h }
    VAR { card { x . regular x ∧ x ≤ −− h } }
    DO e := minarc h;
      IF e ≤ −forest-components f THEN
        f := (f ∩ −(top * e * fT*)) ⊔ (f ∩ top * e * fT*)T ⊔ e
      ELSE
        SKIP
      FI;
      h := h ∩ −e ∩ −eT
    OD
  [ spanning-forest f g ]
apply vcg-tc-simp
using kruskal-vc-1 apply simp
using kruskal-vc-2 apply blast
using kruskal-spanning-invariant-def by auto

Because we have shown total correctness, we conclude that a spanning tree exists.

lemma kruskal-exists-spanning:
  symmetric g ⇒ ∃f . spanning-forest f g
using tc-extract-function kruskal-spanning by blast

This implies that a minimum spanning tree exists, which is used in the subsequent correctness proof.
lemma kruskal-exists-minimal-spanning:
assumes symmetric g
shows \( \exists f . \text{minimum-spanning-forest } f g \)
proof
  let \( \{ f . \text{spanning-forest } f g \} \)
  have \( \exists m \in \{ s . \forall z \in \{ s . \text{sum } (m \cap g) \leq \text{sum } (z \cap g) \} \) using finite-regular spanning-forest-def apply simp
using assms kruskal-exists-spanning apply simp
thus \( \exists ?s \) using minimum-spanning-forest-def by simp
qed

Kruskal’s minimum spanning tree algorithm terminates and is correct. This is the same algorithm that is used in the previous correctness proof, with the same precondition and variant, but with a different invariant and postcondition.

theorem kruskal:
VARS e f h
| symmetric g |
f := bot;
h := g;
WHILE h \neq bot
INV \{ kruskal-invariant f g h \}
VAR \{ card \{ x . \text{regular } x \land x \leq \langle \} \}
DO e := minarc h;
  IF e \leq \text{forest-components } f \THEN
  f := (f \cap -(top * e * f^T)) \sqcup (f \cap top * e * f^T) \sqcup e
  ELSE
  SKIP
  FI;
  h := h \cap -e \cap -e^T
OD
| minimum-spanning-forest f g |
proof vcg-tc-simp
assume symmetric g
thus kruskal-invariant bot g g
  using kruskal-vc-1 kruskal-exists-minimal-spanning kruskal-invariant-def by simp
next
fix n f h
let \?e = minarc h
let \?f = (f \cap -(top * ?e * f^T)) \sqcup (f \cap top * ?e * f^T) \sqcup ?e
let \?h = h \cap -?e \cap -?e^T
let \?F = forest-components f
let \?n1 = card \{ x . \text{regular } x \land x \leq \langle \}
let \?n2 = card \{ x . \text{regular } x \land x \leq \langle h \land x \leq -?e \land x \leq -?e^T \}
assume 1: kruskal-invariant f g h \land h \neq bot \land \?n1 = n
from 1 obtain \( w \) where 2: minimum-spanning-forest \( w \vdash f \leq w \sqcup w^T \)
using kruskal-invariant-def by auto
hence 3: regular \( f \) \& regular \( w \) \& regular \( ?e \)
using 1 by (metis kruskal-invariant-def kruskal-spanning-invariant-def
minimum-spanning-forest-def spanning-forest-def minarc-regular)
show \( (?e \leq -?F \iff \text{kruskal-invariant } ?f \vdash \nexists ?h \land ?n2 < n) \land \neg (?e \leq -?F \iff \text{kruskal-invariant } f \vdash \nexists ?h \land ?n2 < n) \)
proof (rule conjI)
  show \( ?e \leq -?F \iff \text{kruskal-invariant } ?f \vdash \nexists ?h \land ?n2 < n \)
  proof
    assume 4: \( ?e \leq -?F \)
    have 5: equivalence \( ?F \)
    using 1 kruskal-invariant-def kruskal-spanning-invariant-def
forest-components-equivalence spanning-forest-def by simp
    have \( ?e^T \ast \top \ast ?e^T = ?e^T \)
    using 1 by (simp add: arc-top-arc minarc-arc)
    hence \( ?e^T \ast \top \ast ?e^T \leq -?F \)
    using 4 5 conv-complement conv-isotone by fastforce
    hence 6: \( ?e \ast ?F \ast ?e = \bot \)
    using le-bot triple-schroeder-p by simp
    show kruskal-invariant \( ?f \vdash \nexists ?h \land ?n2 < n \)
    proof (unfold kruskal-invariant-def, intro conjI)
      show kruskal-spanning-invariant \( ?f \vdash \nexists ?h \)
      using 1 4 kruskal-vc-2 kruskal-invariant-def by simp
    next
    show \( \exists w . \text{minimum-spanning-forest } w \vdash ?f \leq w \sqcup w^T \)
    proof
      let \( ?p = w \sqcap \top \ast ?e \ast w^T \ast \)
      let \( ?v = (w \sqcap -(\top \ast ?e \ast w^T \ast)) \sqcup ?p^T \)
      have 7: regular \( ?p \)
      using 3 regular-closed-star regular-cone-closed regular-mult-closed by simp
      have 8: injective \( ?v \)
      apply (rule kruskal-exchange-injective-inv-1)
      using 2 minimum-spanning-forest-def spanning-forest-def apply simp
      apply (simp add: covector-mult-closed)
      apply (simp add: comp-associative comp-isotone
star.right-plus-below-circ)
      using 1 2 kruskal-injective-inv-3 minarc-arc
minimum-spanning-forest-def spanning-forest-def by simp
      have 9: components \( g \leq \text{forest-components } ?v \)
      apply (rule kruskal-exchange-spanning-inv-1)
      using 8 apply simp
      using 7 apply simp
      using 2 minimum-spanning-forest-def spanning-forest-def by simp
      have 10: spanning-forest \( ?v \vdash g \)
      proof (unfold spanning-forest-def, intro conjI)
        show injective \( ?v \)
        using 8 by simp

81
next
show acyclic ?v
  apply (rule kruskal-exchange-acyclic-inv-1)
  using 2 minimum-spanning-forest-def spanning-forest-def
        apply simp
        by (simp add: covector-mull-closed)
next
show ?v ≤ −− g
  apply (rule sup-least)
  using 2 inf.coboundedI1 minimum-spanning-forest-def
spanning-forest-def
apply simp
  using 1 2 by (metis kruskal-invariant-def kruskal-spanning-invariant-def
conv-complement conv-dist-inf order.trans
inf.absorb2 inf.coboundedI1 minimum-spanning-forest-def spanning-forest-def)
next
show components g ≤ forest-components ?v
  using 9 by simp
next
show regular ?v
  using 3 regular-closed-star regular-conv-closed regular-mult-closed
  by simp
qed
have 11: sum (?v ⊓ g) = sum (w ⊓ g)
proof −
  have sum (?v ⊓ g) = sum (w ⊓ -(top * ?e * w^T*) ⊓ g) + sum (?p^T ⊓ g)
  using 2 by (metis conv-complement conv-top epm-8 inf-import-p
inf-top-right regular-closed-top vector-top-closed minimum-spanning-forest-def
spanning-forest-def sum-disjoint)
  also have ... = sum ((w ⊓ -(top * ?e * w^T*)) ⊓ g) + sum (?p ⊓ g)
  using 1 kruskal-invariant-def kruskal-spanning-invariant-def
  sum-symmetric
  by simp
  also have ... = sum (((w ⊓ -(top * ?e * w^T*)) ⊓ ?p) ⊓ g)
  using inf-commute inf-left-commute sum-disjoint
  by simp
  also have ... = sum (w ⊓ g)
  using 3 7 maddux-3-11-pp
finally show ?thesis
  by simp
qed
have 12: ?v ∪ ?v^T = w ∪ w^T
proof −
  using conv-complement conv-dist-inf conv-dist-sap inf-import-p
  sup-associative
  by simp
  also have ... = w ∪ w^T
  using 3 7 conv-complement conv-dist-inf inf-import-p maddux-3-11-pp
  sup-monoid.add-associative sup-monoid.add-commute
  by simp
finally show ?thesis
  by simp
qed
have 13: \( ?v * ?e^T = \text{bot} \)
apply (rule kruskal-reroot-edge)
using 1 apply (simp add: minarc-arc)
using 2 minimum-spanning-forest-def spanning-forest-def by simp
have \( ?v \cap ?e \leq ?v \cap \text{top} * ?e \)
using inf.sup-right-isotone top-left-mult-increasing by simp
also have \( \ldots \leq ?v * (\text{top} * ?e)^T \)
using covector-restrict-comp-conv covector-mult-closed vector-top-closed
by simp
finally have 14: \( ?v \cap ?e = \text{bot} \)
using 13 by (metis conv-dist-comp mult-assoc le-bot mult-left-zero)
let \( ?d = ?v \cap \text{top} * ?e^T * ?e^T \cap \text{top} * \text{top} * ?e * -?F \)
let \( ?w = (?v \cap -?d) \sqcup ?e \)
have 15: regular \( ?d \)
using 3 regular-closed-star regular-conv-closed regular-mult-closed by simp
have 16: \( ?F \leq -?d \)
apply (rule kruskal-edge-between-components-1)
using 5 apply simp
using 1 conv-dist-comp minarc-arc mult-associative by simp
have 17: \( f \sqcup f^T \leq (?v \cap -?d \cap -?d^T) \sqcup (?v \cap -?d \cap -?d^T) \)
apply (rule kruskal-edge-between-components-2)
using 16 apply simp
using 1 kruskal-invariant-def kruskal-spanning-invariant-def spanning-forest-def apply simp
using 2 12 by (metis conv-dist-sup conv-involutive conv-isotone le-supI sup-commute)
show minimum-spanning-forest \( ?w \sqcap \text{top} \leq \text{inf} \leq \text{bot} \)
proof (intro conjI)
have 18: \( ?e^T \leq ?v^* \)
apply (rule kruskal-edge-arc-1[where \( g=g \) and \( h=h \)])
using minarc-below apply simp
using 1 apply (metis kruskal-invariant-def kruskal-spanning-invariant-def inf-le1)
using 1 kruskal-invariant-def kruskal-spanning-invariant-def apply simp
using 9 apply simp
using 13 by simp
have 19: arc \( ?d \)
apply (rule kruskal-edge-arc)
using 5 apply simp
using 10 spanning-forest-def apply blast
using 1 apply (simp add: minarc-arc)
using 3 apply (metis conv-complement pp-dist-star regular-mult-closed)
using 2 8 12 apply (simp add: kruskal-forest-components-inf)
using 10 spanning-forest-def apply simp
using 13 apply simp
using 6 apply simp

83
using 18 by simp
show minimum-spanning-forest ?w g
proof (unfold minimum-spanning-forest-def, intro conjI)
have (?v ∩ −?d) * ?eT ≤ ?v * ?eT
  using inf-le1 mult-left-isotone by simp
hence (?v ∩ −?d) * ?eT = bot
  using 13 le-bot by simp
hence 20: ?e * (?v ∩ −?d)T = bot
  using conv-dist-comp cone-involutive cone-bot by force
have 21: injective ?w
  apply (rule injective-sup)
  using 8 apply (simp add: injective-inf-closed)
  using 20 apply simp
  using 1 arc-injective minarc-arc by blast
show spanning-forest ?w g
proof (unfold spanning-forest-def, intro conjI)
  show injective ?w
    using 21 by simp
next
  show acyclic ?w
    apply (rule kruskal-exchange-acyclic-inv-2)
    using 10 spanning-forest-def apply blast
  using 8 apply simp
  using inf.coboundedI1 apply simp
  using 19 apply simp
  using 1 apply (simp add: minarc-arc)
  using inf.cobounded2 inf.coboundedI1 apply simp
  using 13 by simp
next
  have ?w ≤ ?v ∪ ?e
    using inf-le1 sup-left-isotone by simp
  also have ... ≤ −−g ∪ −−h
    by (simp add: le-supI2 minarc-below)
  also have ... = −−g
    using 1 by (metis kruskal-invariant-def
    kruskal-spanning-invariant-def pp-isotone-inf sup.orderE)
  finally show ?w ≤ −−g
    by simp
next
  have 22: ?d ≤ (?v ∩ −?d)T * ?eT * top
    apply (rule kruskal-exchange-spanning-inv-2)
    using 8 apply simp
    using 13 apply (metis semiring.mult-not-zero star-absorb
    star-simulation-right-equal)
    using 17 apply simp
    by (simp add: inf.coboundedI1)
  have components g ≤ forest-components ?v

84
using 10 spanning-forest-def by auto
also have ... ≤ forest-components ?w
  apply (rule kruskal-exchange-forest-components-inv)
using 21 apply simp
using 15 apply simp
using 1 apply (simp add: arc-top-arc minarc-arc)
apply (simp add: inf.coboundedI1)
using 13 apply simp
using 8 apply simp
apply (simp add: le-infI1)
using 22 by simp
finally show components g ≤ forest-components ?w
  by simp
next
  show regular ?w
    using 3 7 regular-conv-closed by simp
qed
next
  have 23: ?e ∩ g ≠ bot
    using 1 by (metis kruskal-invariant-def
      kruskal-spanning-invariant-def comp-inf. semiring.mult-zero-right
      inf.sup-monoid.add-assoc inf.sup-monoid.add-commute minarc-bot-iff
      minarc-meet-bot)
    have g ∩ −h ≤ (g ∩ −h)*
      using star.circ-increasing by simp
    also have ... ≤ (−−(g ∩ −h))*
      using pp-increasing star-isotone by blast
    also have ... ≤ ?F
      using 1 kruskal-invariant-def kruskal-spanning-invariant-def
      inf.sup-monoid.add-commute spanning-forest-def by simp
    finally have 24: g ∩ −h ≤ ?F
      by simp
    have ?d ≤ −−g
      using 10 inf.coboundedI1 spanning-forest-def by blast
    hence ?d ≤ −−g ∩ −?F
      using 16 inf.boundedI p-antitone-iff by simp
    also have ... = −−(g ∩ −?F)
      by simp
    also have ... ≤ −−h
      using 24 p-shunting-swap pp-isotone by fastforce
    finally have 25: ?d ≤ −−h
      by simp
    have ?d = bot −→ top = bot
      using 19 by (metis mult-left-zero mult-right-zero)
    hence ?d ≠ bot
      using 1 le-bot by auto
    hence 26: ?d ∩ h ≠ bot
      using 25 by (metis inf.absorb-iff2 inf-commute pseudo-complement)
    have sum (?e ∩ g) = sum (?e ∩ −−h ∩ g)
85
by (simp add: inf.absorb1 minarc-below)
also have ... = sum (?e ∩ h)
using 1 by (metis kruskal-inv-def
kruskal-spanning-inv-def inf.left-commute inf.sup-monoid.add-commute)
also have ... ≤ sum (?d ∩ h)
using 19 26 minarc-min by simp
also have ... = sum (?d ∩ (¬h ∩ g))
using 1 kruskal-inv-def kruskal-spanning-inv-def
inf-commute by simp
also have ... = sum (?e ∩ h)
using 1 kruskal-inv-def kruskal-spanning-inv-def
inf-commute by simp
also have ... = sum (?d ∩ g)
using 25 by (simp add: inf.absorb2 inf-assoc inf-commute)
finally have 27: sum (?e ∩ g) ≤ sum (?d ∩ g)
by simp
have ?v ∩ ?e ∩ ¬?d = bot
using 14 by simp
hence sum (?w ∩ g) = sum (?v ∩ ¬?d ∩ g) + sum (?e ∩ g)
using sum-disjoint inf-commute inf-assoc by simp
also have ... ≤ sum (?v ∩ ¬?d ∩ g) + sum (?d ∩ g)
using 23 27 sum-plus-right-isotone by simp
also have ... = sum (((?v ∩ ¬?d) ∪ ?d) ∩ g)
using sum-disjoint inf-le2 pseudo-complement by simp
also have ... = sum ((?v ∪ ?d) ∩ (¬?d ∪ ?d) ∩ g)
by (simp add: sup-inf-distrib2)
also have ... = sum ((?v ∪ ?d) ∩ g)
using 15 by (metis inf-top-right stone)
also have ... = sum (?v ∩ g)
by (simp add: inf.sup-monoid.add-assoc)
finally have sum (?w ∩ g) ≤ sum (?v ∩ g)
by simp
thus ∀ u. spanning-forest u g → sum (?w ∩ g) ≤ sum (u ∩ g)
using 2 11 minimum-spanning-forest-def by auto
qed
next
have ?f ≤ f ∪ fT ∪ ?e
using conv-dist-inf inf-le1 sup-left-isotope sup-mono by presburger
also have ... ≤ (?v ∩ ¬?d ∩ ¬?dT) ∪ (¬?vT ∩ ¬?d ∩ ¬?dT) ∪ ?e
using 17 sup-left-isotope by simp
also have ... ≤ (?v ∩ ¬?d) ∪ (?vT ∩ ¬?d ∩ ¬?dT) ∪ ?e
using inf.cobounded1 sup-inf-distrib2 by presburger
also have ... = ?w ∪ (?vT ∩ ¬?d ∩ ¬?dT)
by (simp add: sup-assoc sup-commute)
also have ... ≤ ?w ∪ (?vT ∩ ¬?dT)
using inf.sup-right-isotope inf-assoc sup-right-isotope by simp
also have ... ≤ ?w ∪ ?wT
by simp
using conv-complement conv-dist-inf conv-dist-sup sup-right-isotope
by simp
finally show ?f ≤ ?w ∪ ?wT
by simp
qed
qed

next
show \( ?n^2 < n \)
using 1 kruskal-vc-2 kruskal-invariant-def by auto
qed

next
show \( \neg ?e \leq -?F \longrightarrow \text{kruskal-invariant } fg \ ?h \land \ ?n^2 < n \)
using 1 kruskal-vc-2 kruskal-invariant-def by auto
qed

next
fix \( f \ g \ h \)
assume 28: kruskal-invariant \( f \ g \ h \land h = \bot \)
hence 29: spanning-forest \( f \ g \)
using kruskal-invariant-def kruskal-spanning-invariant-def by auto
from 28 obtain \( w \) where 30: minimum-spanning-forest \( w \ g \ \land f \leq w \sqcup w^T \)
using kruskal-invariant-def by auto
hence \( w = w \cap -g \)
by (simp add: inf.absorb1 minimum-spanning-forest-def spanning-forest-def)
also have \( \dots \leq w \cap \text{components } g \)
by (metis inf.sup-right-isotone star.circ-increasing)
also have \( \dots \leq w \cap f^* \times f^* \)
using 29 spanning-forest-def inf.sup-right-isotone by simp
also have \( \dots \leq f \sqcup f^T \)
apply (rule cancel-separate-6|where \( z=w \) and \( y=w^T \))
using 30 minimum-spanning-forest-def spanning-forest-def apply simp
using 30 apply (metis conv-dist-inf conv-dist-sup conv-involutive inf.cobounded2 inf.orderE)
using 30 apply (simp add: sup-commute)
using 30 minimum-spanning-forest-def spanning-forest-def apply simp
using 30 by (metis acyclic-star-below-complement comp-inf addition.left)+ inf.orderE)
finally have 31: \( w \leq f \sqcup f^T \)
by simp
have \( \text{sum } (f \cap g) = \text{sum } ((w \sqcup w^T) \cap (f \cap g)) \)
using 30 by (metis inf.absorb2 inf.assoc)
also have \( \dots = \text{sum } (w \cap (f \cap g)) + \text{sum } (w^T \cap (f \cap g)) \)
using 30 inf.commute acyclic-asymmetric sum-disjoint
minimum-spanning-forest-def spanning-forest-def by simp
also have \( \dots = \text{sum } (w \cap (f \cap g)) + \text{sum } (w \cap (f^T \cap g)) \)
by (metis conv-dist-inf conv-injective sum-cone)
also have \( \dots = \text{sum } (f \cap (w \cap g)) + \text{sum } (f^T \cap (w \cap g)) \)
using 28 inf.commute inf.assoc kruskal-invariant-def
kruskal-spanning-invariant-def by simp
also have \( \dots = \text{sum } ((f \sqcup f^T) \cap (w \cap g)) \)
using 29 acyclic-asymmetric inf.sup-monoid.add-commute sum-disjoint
spanning-forest-def by simp
also have \( \dots = \text{sum } (w \cap g) \)
using 31 by (metis inf.absorb2 inf.assoc)
finally show minimum-spanning-forest f g
  using 29 30 minimum-spanning-forest-def by simp
qed

8.2 Prim’s Minimum Spanning Tree Algorithm

The total-correctness proof of Prim’s minimum spanning tree algorithm has
the same overall structure as the proof of Kruskal’s algorithm. The partial-
correctness proof is discussed in [1, 4].

abbreviation component g r ≡ r ⊓−− g

definition spanning-tree t g r ≡ forest t ∧ t ≤ (component g r)T ∧ (component g r) □ −− g ∧ component g r ≤ rT * t* ∧ regular t

definition minimum-spanning-tree t g r ≡ spanning-tree t g r ∧ (∀ u . spanning-tree u g r −→ sum (t ⊓ g) ≤ sum (u ⊓ g))

definition prim-precondition g r ≡ g = g ∧ injective r ∧ vector r ∧ regular r

definition prim-spanning-invariant t v g r ≡ prim-precondition g r ∧ vT = rT * t* ∧ spanning-tree t (v * vT ⊓ g) r

definition prim-invariant t v g r ≡ prim-spanning-invariant t v g r ∧ (∃ w . minimum-spanning-tree w g r ∧ t ≤ w)

lemma span-tree-split:
  assumes vector r
  shows t ≤ (component g r)T * (component g r) □ −− g −→ (t ≤ (component g r)T ∧ t ≤ component g r ∧ t ≤ −− g)
proof –
  have (component g r)T * (component g r) = (component g r)T □ component g r
    by (metis assms conv-involutive covector-mult-closed vector-conv-covector vector-covector)
  thus ?thesis
    by simp
qed

lemma span-tree-component:
  assumes spanning-tree t g r
  shows component g r = component t r
using assms by (simp add: antisym mult-right-isotone star-isotone spanning-tree-def)

We first show three verification conditions which are used in both cor-
correctness proofs.

lemma prim-vc-1:
  assumes prim-precondition g r
  shows prim-spanning-invariant bot r g r
proof (unfold prim-spanning-invariant-def, intro conjI)
  show prim-precondition g r
    using assms by simp
next
  show rT = rT * bot
    by (simp add: star-absorb)
let ?ss = r * r^T ⊓ g

show spanning-tree bot ?ss r
----------
proof (unfold spanning-tree-def, intro conjI)
  show injective bot
  by simp
next
  show acyclic bot
  by simp
next
  show bot ≤ (component ?ss r)^T * (component ?ss r) ⊓ −− ?ss
  by simp
next
  have component ?ss r ≤ component (r * r^T) r
    by (simp add: mult-right-isotone star-isotone)
  also have ... ≤ r^T * 1*
    using assms by (metis inf.eq-iff p-antitone regular-one-closed star-sub-one
  
also have ... = r^T * bot*
    by (simp add: star.circ-zero star-one)
  finally show component ?ss r ≤ r^T * bot*
    .
  next
  show regular bot
  by simp
qed

lemma prim-vc-2:
assumes prim-spanning-invariant t v g r
    and v * −v^T ⊓ g ≠ bot
    and card { x . regular x ∧ x ≤ component g r ∧ x ≤ −v^T } = n
    shows prim-spanning-invariant (t ⊔ minarc (v * −v^T ⊓ g)) (v ⊔ minarc (v * −v^T ⊓ g)^T * top) g r ∧ card { x . regular x ∧ x ≤ component g r ∧ x ≤ −(v ⊔ 
    minarc (v * −v^T ⊓ g)^T * top)^T } < n
proof

let ?vcv = v * −v^T ⊓ g
let ?e = minarc ?vcv
let ?t = t ⊔ ?e
let ?v = v ⊔ ?e^T * top
let ?c = component g r
let ?g = −g
let ?n1 = card { x . regular x ∧ x ≤ ?c ∧ x ≤ −v^T }
let ?n2 = card { x . regular x ∧ x ≤ ?c ∧ x ≤ −?v^T }

have 1: regular v ∧ regular (v * v^T) ∧ regular (?v * ?v^T) ∧ regular (top * ?e)
  using assms(1) by (metis prim-spanning-invariant-def spanning-tree-def
  prim-precondition-def regular-conv-closed regular-closed-star regular-mult-closed
  con-inv-star linear-star-close star-involutive regular-closed-star regular-closed-sup
  minarc regular)

hence 2: ?t ≤ v * v^T ⊓ ?g
using \textit{assms}(1) \textit{by} (metis \textit{prim-spanning-invariant-def} \textit{spanning-tree-def} \\
\textit{inf-pp-commute} \textit{inf.boundedE})

hence 3: $t \leq v \ast v^T$ \\
by simp

have 4: $t \leq \varphi g$ \\
using 2 \textit{by} simp

have 5: $\varphi e \leq v \ast -v^T \cap \varphi g$ \\
using 1 \textit{by} (metis \textit{minarc-below} \textit{pp-dist-inf} \textit{regular-mult-closed} \\
\textit{regular-closed-p})

hence 6: $\varphi e \leq v \ast -v^T$ \\
by simp

have 7: vector $v$ \\
using \textit{assms}(1) \textit{prim-spanning-invariant-def} \textit{prim-precondition-def} \textit{by} (simp add: \textit{covector-mult-closed} \textit{vector-covector})

hence 8: $\varphi e \leq v$ \\
using 6 \textit{by} (metis \textit{conv-complement} \textit{inf.boundedE} \textit{vector-complement-closed} \\
\textit{vector-covector})

have 9: $\varphi e \ast t = \text{bot}$ \\
using 3 6 7 et(1) \textit{by} blast

have 10: $\varphi e \ast t^T = \text{bot}$ \\
using 3 6 7 et(2) \textit{by} simp

have 11: arc $\varphi e$ \\
using \textit{assms}(2) \textit{minarc-arc} \textit{by} simp

have $r^T \leq r^T \ast t^*$ \\
by (metis \textit{mult-right-isotone} \textit{order-refl} \textit{semiring.mult-not-zero} \\
\textit{star.circ-separate-mult-1} \textit{star-absorb})

hence 12: $r^T \leq v^T$

using \textit{assms}(1) \textit{by} (simp add: \textit{prim-spanning-invariant-def})

have 13: vector $r \wedge \text{injective} r \wedge v^T = r^T \ast t^*$

using \textit{assms}(1) \textit{prim-spanning-invariant-def} \textit{prim-precondition-def} \\
minimum-spanning-tree-def \textit{spanning-tree-def} \textit{reachable-restrict} \textit{by} simp

have $g = g^T$

using \textit{assms}(1) \textit{prim-invariant-def} \textit{prim-spanning-invariant-def} \\
\textit{prim-precondition-def} \textit{by} simp

hence 14: $g^T = \varphi g$ \\
using \textit{conv-complement} \textit{by} simp

show \textit{prim-spanning-invariant} ?t \ ?v g r \ ?n2 < n

\textit{proof} (rule conjI)

\textit{show} \textit{prim-spanning-invariant} ?t \ ?v g r

\textit{proof} (unfold \textit{prim-spanning-invariant-def}, intro conjI)

\textit{show} \textit{prim-precondition} g r

\textit{using} \textit{assms}(1) \textit{prim-spanning-invariant-def} \textit{by} simp

next

\textit{show} ?v^T = r^T \ast ?t^*

\textit{using} \textit{assms}(1) 6 7 9 \textit{by} (simp add: \textit{reachable-inv} \\
\textit{prim-spanning-invariant-def} \textit{prim-precondition-def} \textit{spanning-tree-def})

next

\textit{let} ?G = ?v \ast ?v^T \cap g

\textit{show} \textit{spanning-tree} ?t ?G r

90
proof (unfold spanning-tree-def, intro conjI)
  show injective ?t
    using assms(1) 10 11 by (simp add: injective-inv
                         prim-spanning-invariant-def spanning-tree-def)
  next
  show acyclic ?t
    using assms(1) 3 6 7 acyclic-inv prim-spanning-invariant-def
    spanning-tree-def by simp
  next
  show ?t ≤ (component ?G r) * (component ?G r) ∩ --- ?G
    using 1 2 5 7 13 prim-subgraph-inv inf-pp-commute mst-subgraph-inv-2
    by auto
  next
  show component (?v * ?vT ∩ g) r ≤ r * ?t
    proof
    have 15: r * (v * vT ∩ ?g) ≤ r * t
      using assms(1) 1 by (metis prim-spanning-invariant-def
                         spanning-tree-def inf-pp-commute)
    have component (?v * ?vT ∩ g) r = r * (?v * ?vT ∩ ?g)
      using 1 by simp
    also have ... ≤ r * ?t
      using 2 6 7 11 12 13 14 15 by (metis span-inv)
    finally show ?thesis
  qed
next
  show regular ?t
    using assms(1) by (metis prim-spanning-invariant-def spanning-tree-def
                         regular-closed-sup minarc-regular)
  qed
  qed
next
  have 16: top * ?e ≤ ?c
    proof
    have top * ?e = top * ?eT * ?e
      using 11 by (metis arc-top-edge mult-assoc)
    also have ... ≤ vT * ?e
      using 7 8 by (metis conv-dist-comp conv-isotone mult-left-isotone
                         symmetric-top-closed)
    also have ... ≤ vT * ?g
      using 5 mult-right-isotone by auto
    also have ... = rT * tT * ?g
      using 13 by simp
    also have ... ≤ rT * ?g * ?g
      using 4 by (simp add: mult-right-isotone mult-right-isotone star-isotone)
    also have ... ≤ ?c
      by (simp add: comp-associative mult-right-isotone star.right-plus-below-circ)
    finally show ?thesis
    by simp
  qed

91
have 17: top * ?e ≤ −v^T

using 6 7 by (simp add: Schroeder-4-p vTeT)

have 18: top * ?e ≤ −(top * ?e)
by (metis assms(2) inf.orderE minarc-bot-iff conv-complement-sub-inf inf-p
inf-top.lef-neutral p-bot symmetric-top-closed vector-top-closed)

have 19: −?v^T = −v^T ⊓ −(top * ?e)
by (simp add: inf-sup-bot-iff)

have 20: ¬top * ?e ≤ −?v^T

using 18 by simp

have ?n2 < ?n1
apply (rule psubset_card_mono)
using finite-regular apply simp
using 1 16 17 19 20 by auto

thus ?n2 < n
using assms(3) by simp

end

lemma prim-vc-3:
assumes prim-spanning-invariant t v g r
and v * −v^T ⊓ g = bot
shows spanning-tree t g r

proof
let ?g = −−g

have 1: regular v ∧ regular (v * v^T)
using assms(1)
by (metis prim-spanning-invariant-def spanning-tree-def
prim-precondition-def regular-conv-closed regular-closed-star regular-mult-closed
conv-involutive)

have 2: v * −v^T ⊖ ?g = bot
using assms(2)
by (simp add: pp-inf-bot-iff)

have 3: v'^T = r^T * t^* ∧ vector v
using assms(1)
by (simp add: covector-mult-closed prim-invariant-def
prim-spanning-invariant-def vector-covector prim-precondition-def)

have 4: t ≤ v * v^T ⊖ ?g
using assms(1) 1 by (metis prim-spanning-invariant-def inf-pp-commute
spanning-tree-def inf boundedE)

have r^T * (v * v^T ⊖ ?g) ⊖ r^T * t^*
using assms(1) 1 by (metis span-post)

have ?n2 < ?n1
by (metis assms(3) prim-spanning-invariant-def spanning-tree-def
inf pp-commute inf boundedE)

92
show \text{spanning-tree } t \ g \ r
\begin{align*}
\text{apply } & (\text{unfold spanning-tree-def, intro conj}) \\
\text{using } & \text{assms(1) prim-spanning-invariant-def spanning-tree-def apply simp} \\
\text{using } & \text{assms(1) prim-spanning-invariant-def spanning-tree-def apply simp} \\
\text{using } & 5 \ 6 \ \text{apply simp} \\
\text{using } & \text{assms(1) 5 prim-spanning-invariant-def apply simp} \\
\text{using } & \text{assms(1) prim-spanning-invariant-def spanning-tree-def by simp}
\end{align*}
qed

The following result shows that Prim’s algorithm terminates and constructs a spanning tree. We cannot yet show that this is a minimum spanning tree.

\textbf{theorem} prim-spanning:
\begin{align*}
\text{VARS } & t \ v \ e \\
\text{prim-precondition } & g \ r \\
 t & := \text{bot}; \\
v & := r; \\
\text{WHILE } & v \ast -v^T \cap g \neq \text{bot} \\
\text{INV } & \{ \text{prim-spanning-invariant } t \ v \ g \ r \} \\
\text{VAR } & \{ \text{card } \{ \ x \ . \ \text{regular } x \land x \leq \text{component } g \cap -v^T \} \} \\
\text{DO } & e := \text{minarc } (v \ast -v^T \cap g); \\
 & t := t \sqcup e; \\
 & v := v \sqcup e^T \ast \text{top} \\
\text{OD} \\
\text{spanning-tree } & t \ g \ r \\
\text{apply } & \text{vcg-tc-simp} \\
\text{apply } & (\text{simp add: prim-vc-1}) \\
\text{using } & \text{prim-vc-2 apply blast} \\
\text{using } & \text{prim-vc-3 by auto}
\end{align*}

Because we have shown total correctness, we conclude that a spanning tree exists.

\textbf{lemma} prim-exists-spanning:
\begin{align*}
\text{prim-precondition } & g \ r \Rightarrow \exists t . \text{spanning-tree } t \ g \ r \\
\text{using } & \text{tc-extract-function prim-spanning by blast}
\end{align*}

This implies that a minimum spanning tree exists, which is used in the subsequent correctness proof.

\textbf{lemma} prim-exists-minimal-spanning:
\begin{align*}
\text{assumes } & \text{prim-precondition } g \ r \\
\text{shows } & \exists t . \text{minimum-spanning-tree } t \ g \ r \\
\text{proof } & - \\
\text{let } & ?s = \{ t . \text{spanning-tree } t \ g \ r \} \\
\text{have } & \exists m \in ?s . \forall z \in ?s . \text{sum } (m \cap g) \leq \text{sum } (z \cap g) \\
\text{apply } & (\text{rule finite-set-minimal}) \\
\text{using } & \text{finite-regular spanning-tree-def apply simp} \\
\text{using } & \text{assms prim-exists-spanning apply simp} \\
\text{using } & \text{sum-linear by simp} \\
\text{thus } & ?\text{thesis}
\end{align*}

93
using \texttt{minimum-spanning-tree-def} by simp

\textit{qed}

Prim’s minimum spanning tree algorithm terminates and is correct. This is the same algorithm that is used in the previous correctness proof, with the same precondition and variant, but with a different invariant and post-condition.

\textbf{theorem} \texttt{prim}:

\begin{verbatim}
VARS t v e
[ prim-precondition g r \land (\exists w . \textit{minimum-spanning-tree} w g r) ]
t := bot;
v := r;
WHILE v \neq v^T \land g \neq bot
INV \{ prim-invariant t v g r \}
VAR \{ card \{ x . \text{regular} x \land x \leq \text{component} g r \land -v^T \} \}
DO e := \text{minarc} (v \neq v^T \land g);
(t := t \sqcup e; 
v := v \sqcup e^T \ast \text{top})
OD
[ \textit{minimum-spanning-tree} t g r ]
\end{verbatim}

\textbf{proof} \texttt{vcg-tc-simp}

\begin{verbatim}
assume prim-precondition g r \land (\exists w . \textit{minimum-spanning-tree} w g r)
thus prim-invariant bot r g r
using prim-invariant-def prim-vc-1 by simp
\end{verbatim}

\textbf{next}

\begin{verbatim}
fix t v n
let \texttt{?vcv} = v \neq v^T \land g
let \texttt{?vv} = v \neq v^T \land g
let \texttt{?e} = \text{minarc} ?vcv
let \texttt{?t} = t \sqcup ?e
let \texttt{?v} = v \sqcup ?e^T \ast \text{top}
let \texttt{?c} = \text{component} g r
let \texttt{?g} = g
let \texttt{?n1} = card \{ x . \text{regular} x \land x \leq ?c \land x \leq -v^T \}
let \texttt{?n2} = card \{ x . \text{regular} x \land x \leq ?c \land x \leq -?v^T \}
assume 1: prim-invariant t v g r \land ?vcv \neq bot \land ?n1 = n
hence 2: regular v \land \text{regular} (v \ast v^T)
by (metis \texttt{(no-types, hide-lams) prim-invariant-def} \texttt{prim-spanning-invariant-def} \texttt{spanning-tree-def} \texttt{prim-precondition-def} \texttt{regular-conv-closed regular-star regular-mult-closed conv-involutive})
have 3: t \leq v \ast v^T \land ?g
using 1 2 by (metis \texttt{(no-types, hide-lams) prim-invariant-def} \texttt{prim-spanning-invariant-def spanning-tree-def inf-pp-commute inf.boundedE})
\textit{hence} 4: t \leq v \ast v^T
by simp
\textit{have} 5: t \leq ?g
using 3 by simp
have 6: ?e \leq v \neq v^T \land ?g
using 2 by (metis minarc-below pp-dist-inf regular-mult-closed)
\end{verbatim}

\textbf{94}
regular-closed-p)

hence 7: \( ?e \leq v \ast -v^T \)
by simp

have 8: vector \( v \)
using 1 prim-invariant-def prim-spanning-invariant-def prim-precondition-def
by (simp add: covector-mult-closed vector-conv-covector)

have 9: arc \( ?e \)
using 1 minarc-arc by simp

from 1 obtain \( w \) where 10: minimum-spanning-tree \( w \) \( g \) \( r \) \( t \) \( \leq w \)
by (metis prim-invariant-def)

hence 11: vector \( r \) \& injective \( r \) \& \( v^T = r^T \ast t^* \) \& forest \( w \) \& \( t \) \( \leq w \) \& \( w \) \leq \( \ ?c^T \ast ?e \ \ast \ ?g \ \ast \ ?w^T \ast \ ?c^T \ast ?e \ \ast \ ?g \ast \ ?w^T \ast \ ?e \)
using 1 2 prim-invariant-def prim-spanning-invariant-def
prim-precondition-def minimum-spanning-tree-def spanning-tree-def
reachable-restrict by simp

have 12: \( w \ast v \leq v \)
using predecessors-reachable reachable-restrict by auto

have 13: \( g = g^T \)
using 1 prim-invariant-def prim-spanning-invariant-def prim-precondition-def
by simp

hence 14: \( ?g^T = ?g \)
using conv-complement by simp

show prim-invariant \( ?t \) \( ?v \) \( g \) \( r \) \( ?n2 \leq n \)
proof (unfold prim-invariant-def, intro conjI)
show prim-spanning-invariant \( ?t \) \( ?v \) \( g \) \( r \)
using 1 prim-invariant-def prim-vc-2 by blast

next
show \( \exists w \). minimum-spanning-tree \( w \) \( g \) \( r \) \& \( ?t \leq w \)
proof
let \( ?f = w \ \cap \ v \ast -v^T \ \cap \ top \ast \ ?c \ast \ ?w^T \ast \ ?e \)
let \( ?p = w \ \cap \ -v \ast -v^T \ \cap \ top \ast \ ?c \ast \ ?w^T \ast \ ?e \)
let \( ?fp = w \ \cap \ -v^T \ \cap \ top \ast \ ?c \ast \ ?w^T \ast \ ?e \)
let \( ?w = (w \ \cap \ -?fp) \ \sqcup \ ?p^T \ \sqcup \ ?e \)

have 15: regular \( ?f \) \& regular \( ?fp \) \& regular \( ?w \)
using 2 10 by (metis regular-cone-closed regular-closed-star
regular-mult-closed regular-closed-top regular-closed-inf regular-closed-sup
minarc-regular minimum-spanning-tree-def spanning-tree-def)

show minimum-spanning-tree \( ?w \) \( g \) \( r \) \& \( ?t \leq \ ?w \)
proof (intro conjI)
show minimum-spanning-tree \( ?w \) \( g \) \( r \)
proof (unfold minimum-spanning-tree-def, intro conjI)
show spanning-tree \( ?w \) \( g \) \( r \)
proof (unfold spanning-tree-def, intro conjI)
show injective \( ?w \)
using 7 8 9 11 exchange-injective by blast

next
show acyclic \( ?w \)
using 7 8 11 12 exchange-acyclic by blast
next
show \( w \leq c^T \cap -g \)

proof -

have 16: \( w \cap -fp \leq c^T \cap -g \)
using 10 by (simp add: le-infI1 minimum-spanning-tree-def spanning-tree-def)

have \( \beta^T \leq w^T \)
by (simp add: conv-isotone inf.sup-monoid.add-assoc)

also have \( \ldots \leq (c^T \cap -g)^T \)
using 11 conv-order by simp

also have \( \ldots = c^T \cap -g \)
using 2 14 conv-dist-comp conv-dist-inf by simp

finally have 17: \( \beta^T \leq c^T \cap -g \)

have \( \gamma \leq c^T \cap \gamma \)
using 5 6 11 mst-subgraph-inv by auto

thus ?thesis
using 16 17 by simp

qed

next

show \( c \leq r^T \cap w^* \)

proof -

have \( \gamma \leq r^T \cap w^* \)
using 10 minimum-spanning-tree-def spanning-tree-def by simp

also have \( \ldots \leq r^T \cap w^* \)
using 4 7 8 10 11 12 15 by (metis mst-reachable-inv)

finally show ?thesis

qed

next

show regular \( w \)
using 15 by simp

qed

next

have 18: \( \forall f \Box \forall p = \forall fp \)
using 2 8 epm-1 by fastforce

have \( \text{arc} (w \cap -v \cap -v^T \cap \gamma \leq \gamma \cap w^T) \)
using 5 6 8 9 11 12 reachable-restrict arc-edge by auto

hence 19: \( \text{arc} \ ?f \)
using 2 by simp

hence \( \forall f = \text{bot} \rightarrow \top = \text{bot} \)
by (metis mult-left-zero mult-right-zero)

hence \( \forall f \neq \bot \)
using 1 le-bot by auto

hence \( \forall f \cap v \ast -v^T \cap \gamma \neq \bot \)
using 2 11 by (simp add: inf.absorb1 le-infI1)

hence \( g \cap (\forall f \cap v \ast -v^T) \neq \bot \)
using inf-commute pp-inf-bot-iff by simp

hence 20: \( \forall f \cap \forall v \neq \bot \)
by (simp add: inf-assoc inf-commute)
hence 21: \( ?f \cap g = ?f \cap ?vec \)

using 2 by (simp add: inf-assoc inf-commute inf-left-commute)

have 22: \( ?e \cap g = \text{minarc} \ ?vec \cap ?vec \)

using 7 by (simp add: inf.absorb2 inf.assoc inf.commute)

hence 23: \( \text{sum} (\ ?e \cap g) \leq \text{sum} (\ ?f \cap g) \)

using 15 19 20 21 by (simp add: minarc-min)

have \(?e \neq \text{bot}\)

using 20 comp-inf.semiring.mutl-not-zero semiring.mult-not-zero by blast

hence 24: \(?e \cap g \neq \text{bot}\)

using 22 minarc-meet-bot by auto

have \(\text{sum} (\ ?w \cap g) = \text{sum} (\ w \cap \ ?fp \cap g) + \text{sum} (\ ?pT \cap g) + \text{sum} (\ ?e \cap g)\)

using 7 8 10 by (metis sum-disjoint-3 epm-8 epm-9 epm-10 minimum-spanning-tree-def spanning-tree-def prim-precondition-def regular-conv-closed regular-mult-closed conv-involutive)

also have \(\ldots\) using 11 by (metis sum-plus-right-isotone)

also have \(\ldots\) using 23 24 by (simp add: sum-plus-right-isotone)

also have \(\ldots\) using 11 by (metis epm-8 sum-disjoint)

also have \(\ldots\) using 13 sum-symmetric by auto

also have \(\ldots\) using 2 8 by (metis sum-disjoint-3 epm-11 epm-12 epm-13)

also have \(\ldots\) using 2 8 15 18 epm-2 by force

finally have \(\text{sum} (\ ?w \cap g) \leq \text{sum} (\ w \cap g)\)

thus \(\forall u . \text{spanning-tree} u g r \rightarrow \text{sum} (\ ?w \cap g) \leq \text{sum} (\ u \cap g)\)

using 10 order-lesseq-imp minimum-spanning-tree-def by auto

qed

next

show \(?t \leq ?w\)

using 4 8 10 mst-extends-new-tree by simp

qed

next

show \(?n2 < n\)

using 1 prim-invariant-def prim-vc-2 by auto

qed

next

fix \(t v\)

let \(?g = \neg \neg g\)

assume 25: prim-invariant \(t v g r \land v \ast \neg vT \cap g = \text{bot}\)

hence 26: regular \(v\)

from 25 obtain w where 27: minimum-spanning-tree w g r ∧ t ≤ w by (metis prim-invariant-def)
have spanning-tree t g r using 25 prim-invariant-def prim-vc-3 by blast
hence component g r = vₜ by (metis 25 prim-invariant-def span-tree-component
prim-spanning-invariant-def spanning-tree-def)
hence 28: w ≤ v ∗ vₜ using 26 27 by (simp add: minimum-spanning-tree-def spanning-tree-def
inf-pp-commute)
have vector r ∧ injective r ∧ forest w using 25 27 by (simp add: prim-invariant-def prim-spanning-invariant-def
prim-precondition-def minimum-spanning-tree-def spanning-tree-def)
hence w = t using 25 27 28 prim-invariant-def prim-spanning-invariant-def mst-post by blast
thus minimum-spanning-tree t g r
using 27 by simp
qed
end
end

References


