Aggregation Algebras

Walter Guttmann

March 17, 2025

Abstract

We develop algebras for aggregation and minimisation for weight matrices and for edge weights in graphs. We show numerous instances of these algebras based on linearly ordered commutative semigroups.

Contents

1 Overview

This document describes the following four 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.

Three theory files, which were originally part of this entry, have been moved elsewhere:

* A theory for total-correctness proofs in Hoare logic became part of Isabelle/HOL's theory `Hoare/Hoare_Logic.thy`.

* A theory with simple total-correctness proof examples became Isabelle/HOL's theory `Hoare/ExamplesTC.thy`.

* A theory proving total correctness of Kruskal's and Prim's minimum spanning tree algorithms based on m-Kleene-algebras using Hoare logic was split into two theories that became part of AFP entry [6].

Following a refactoring, the selection of components of graphs in m-Kleene-algebras, which was originally part of Nicolas Robinson-O'Brien's theory `Relational_Minimum_Spanning_Trees/Boruvka.thy`, has been moved into a new theory in this entry.

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 *Main*
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*]: $x * y * \mathbf{1} = x * y$
  assumes *z-idem* [*simp*]: $\mathbf{1} * \mathbf{1} = \mathbf{1}$
begin

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

interpretation *comp?*: *comp-fun-commute* $f \circ g$
  by (*fact comp-comp-fun-commute*)

definition $F :: ('b \Rightarrow 'a) \Rightarrow 'b\ set \Rightarrow 'a$
  where *eq-fold*: $F\ g\ A = \mathit{Finite\text{-}Set.fold}\ (f \circ g)\ \mathbf{1}\ A$

lemma *infinite* [*simp*]: $\neg$ *finite* $A \Longrightarrow F\ g\ A = \mathbf{1}$
  by (*simp add*: *eq-fold*)

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

lemma *insert* [*simp*]: *finite* $A \Longrightarrow x \notin A \Longrightarrow F\ g\ (\mathit{insert}\ x\ A) = g\ x * F\ g\ A$
  by (*simp add*: *eq-fold*)

lemma *remove*:
  assumes *finite* A **and** $x \in A$
  shows $F\ g\ A = g\ x * F\ g\ (A - \{x\})$
proof −
  from ‹$x \in A$› **obtain** B **where** B: $A = \mathit{insert}\ x\ B$ **and** $x \notin B$
    by (*auto dest*: *mk-disjoint-insert*)
  moreover from ‹*finite* A› B **have** *finite* B **by** *simp*
  ultimately show *?thesis* **by** *simp*
qed

lemma *insert-remove*: *finite* $A \Longrightarrow F\ g\ (\mathit{insert}\ x\ A) = g\ x * F\ g\ (A - \{x\})$

by (*cases* $x \in A$) (*simp-all add*: *remove insert-absorb*)

lemma *insert-if*: *finite* $A \Longrightarrow F\ g\ (insert\ x\ A) = (if\ x \in A\ then\ F\ g\ A\ else\ g\ x * F\ g\ A)$
by (*cases* $x \in A$) (*simp-all add*: *insert-absorb*)

lemma *neutral*: $\forall x{\in}A.\ g\ x = \mathbf{1} \Longrightarrow F\ g\ A = \mathbf{1}$
by (*induct A rule*: *infinite-finite-induct*) *simp-all*

lemma *neutral-const* [*simp*]: $F\ (\lambda\text{-}.\ \mathbf{1})\ A = \mathbf{1}$
by (*simp add*: *neutral*)

lemma *F-one* [*simp*]: $F\ g\ A * \mathbf{1} = F\ g\ A$
proof −
have $\bigwedge f\ b\ B.\ F\ f\ (insert\ (b{::}'b)\ B) * \mathbf{1} = F\ f\ (insert\ b\ B) \lor infinite\ B$
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*]: $\mathbf{1} * F\ g\ A = F\ g\ A$
using *F-one commute* **by** *auto*

lemma *F-g-one* [*simp*]: $F\ (\lambda x\ .\ g\ x * \mathbf{1})\ A = F\ g\ A$
apply (*induct A rule*: *infinite-finite-induct*)
apply *simp*
apply *simp*
by (*metis one-F assoc insert*)

lemma *union-inter*:
assumes *finite A* **and** *finite B*
shows $F\ g\ (A \cup B) * F\ g\ (A \cap B) = F\ g\ A * F\ g\ B$
— 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 $\forall x \in A \cap B.\ g\ x = \mathbf{1}$
shows $F\ g\ (A \cup B) = F\ g\ A * F\ g\ B$
using *assms* **by** (*simp add*: *union-inter* [*symmetric*] *neutral*)

corollary *union-disjoint*:
  assumes *finite* A **and** *finite* B
  assumes $A \cap B = \{\}$
  shows $F\ g\ (A \cup B) = F\ g\ A * F\ g\ B$
  using *assms* **by** (*simp add*: *union-inter-neutral*)

lemma *union-diff2*:
  assumes *finite* A **and** *finite* B
  shows $F\ g\ (A \cup B) = F\ g\ (A - B) * F\ g\ (B - A) * F\ g\ (A \cap B)$
proof −
  have $A \cup B = A - B \cup (B - A) \cup A \cap B$
    by *auto*
  with *assms* **show** *?thesis*
    by *simp* (*subst union-disjoint*, *auto*)+
qed

lemma *subset-diff*:
  assumes $B \subseteq A$ **and** *finite* A
  shows $F\ g\ A = F\ g\ (A - B) * F\ g\ B$
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 $F\ g\ (A - B \cup B) = F\ g\ (A - B) * F\ g\ B$ **by** (*rule union-disjoint*)
  moreover from *assms* **have** $A \cup B = A$ **by** *auto*
  ultimately show *?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 $F\ g\ A \neq \mathbf{1}$
  obtains a **where** $a \in A$ **and** $g\ a \neq \mathbf{1}$
proof −
  from *assms* **have** $\exists\, a{\in}A.\ g\ a \neq \mathbf{1}$
  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 a A*)
    then show *?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 ∘ 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: ⋀x. x ∈ B ⟹ g x = h x
  shows F g A = F h B
  using g-h unfolding ‹A = B›
  by (induct B rule: infinite-finite-induct) auto

lemma strong-cong [cong]:
  assumes A = B ⋀x. x ∈ B =simp=> g 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 ⋀x. x ∈ B ⟹ 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 ∀ i∈I. finite (A i)
    and ∀ i∈I. ∀ j∈I. i ≠ j ⟶ A i ∩ A j = {}
  shows F g (⋃(A ‘ I)) = F (λx. F g (A x)) I
  apply (insert assms)
  apply (induct rule: finite-induct)
   apply simp
  apply atomize
  apply (subgoal-tac ∀ i∈F. x ≠ i)
   prefer 2 apply blast
  apply (subgoal-tac A x ∩ ⋃(A ‘ F) = {})
   prefer 2 apply blast
  apply (simp add: union-disjoint)
```

done

lemma *Union-disjoint*:
assumes $\forall A{\in}C.\ finite\ A$ $\forall A{\in}C.\ \forall B{\in}C.\ A \neq B \longrightarrow A \cap B = \{\}$
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*: $F\ (\lambda x.\ g\ x * h\ x)\ A = F\ g\ A * F\ h\ A$
by (*induct A rule*: *infinite-finite-induct*) (*simp-all add*: *assoc commute left-commute*)

lemma *Sigma*:
$finite\ A \Longrightarrow \forall x{\in}A.\ finite\ (B\ x) \Longrightarrow F\ (\lambda x.\ F\ (g\ x)\ (B\ x))\ A = F\ (case\text{-}prod\ g)\ (SIGMA\ x{:}A.\ B\ x)$
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*: $R\ \mathbf{1}\ \mathbf{1}$
and *Rop*: $\forall x1\ y1\ x2\ y2.\ R\ x1\ x2 \wedge R\ y1\ y2 \longrightarrow R\ (x1 * y1)\ (x2 * y2)$
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 $\forall i \in T - S.\ h\ i = \mathbf{1}$
and $\bigwedge x.\ x \in S \Longrightarrow g\ x = h\ x$
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*:
  finite $T \Longrightarrow S \subseteq T \Longrightarrow \forall\, i \in T - S.\ g\ i = \mathbf{1} \Longrightarrow (\bigwedge x.\ x \in S \Longrightarrow g\ x = h\ x) \Longrightarrow$
    $F\ g\ T = F\ h\ S$
  by (*auto intro*!: *mono-neutral-cong-left* [*symmetric*])

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

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

lemma *mono-neutral-cong*:
  assumes [*simp*]: *finite T finite S*
    and $*$: $\bigwedge i.\ i \in T - S \Longrightarrow h\ i = \mathbf{1}$ $\bigwedge i.\ i \in S - T \Longrightarrow g\ i = \mathbf{1}$
    and *gh*: $\bigwedge x.\ x \in S \cap T \Longrightarrow g\ x = h\ x$
 shows $F\ g\ S = F\ h\ T$
proof −
  have $F\ g\ S = F\ g\ (S \cap T)$
    by(*rule mono-neutral-right*)(*auto intro*: $*$)
  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*: $*$)
  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*:
    $\bigwedge a.\ a \in S \Longrightarrow i\ (j\ a) = a$
    $\bigwedge a.\ a \in S \Longrightarrow j\ a \in T$
    $\bigwedge b.\ b \in T \Longrightarrow j\ (i\ b) = b$
    $\bigwedge b.\ b \in T \Longrightarrow i\ b \in S$
  assumes *eq*:
    $\bigwedge a.\ a \in S \Longrightarrow h\ (j\ a) = g\ a$

shows $F\ g\ S = F\ h\ T$
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 $F\ g\ S = F\ (\lambda x.\ h\ (j\ x))\ S$
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* $h\ (S - S')\ (T - T')$
assumes *nn*:
$\bigwedge a.\ a \in S' \Longrightarrow g\ (h\ a) = z$
$\bigwedge b.\ b \in T' \Longrightarrow g\ b = z$
shows $F\ (\lambda x.\ g\ (h\ x))\ S = F\ g\ T$
proof –
have [*simp*]: *finite* $S \longleftrightarrow$ *finite* T
using *bij-betw-finite*[*OF bij*] *fin* **by** *auto*
show *?thesis*
proof (*cases finite* S)
case *True*
with *nn* **have** $F\ (\lambda x.\ g\ (h\ x))\ S = F\ (\lambda x.\ g\ (h\ x))\ (S - S')$
by (*intro mono-neutral-cong-right*) *auto*
also have $\ldots = F\ g\ (T - T')$
using *bij* **by** (*rule reindex-bij-betw*)
also have $\ldots = F\ g\ T$
using *nn* ‹*finite* S› **by** (*intro mono-neutral-cong-left*) *auto*
finally show *?thesis* **.**
next
case *False*
then show *?thesis* **by** *simp*
qed
qed

lemma *reindex-nontrivial*:
assumes *finite* A
and *nz*: $\bigwedge x\ y.\ x \in A \Longrightarrow y \in A \Longrightarrow x \neq y \Longrightarrow h\ x = h\ y \Longrightarrow g\ (h\ x) = \mathbf{1}$
shows $F\ g\ (h\ `\ A) = F\ (g \circ h)\ A$
proof (*subst reindex-bij-betw-not-neutral* [*symmetric*])
show *bij-betw* $h\ (A - \{x \in A.\ (g \circ h)\ x = \mathbf{1}\})\ (h\ `\ A - h\ `\ \{x \in A.\ (g \circ h)\ x = \mathbf{1}\})$
using *nz* **by** (*auto intro*!: *inj-onI simp*: *bij-betw-def*)
qed (*use* ‹*finite* A› **in** *auto*)

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

assumes *witness*:
$\bigwedge a.\ a \in S - S' \Longrightarrow i\ (j\ a) = a$
$\bigwedge a.\ a \in S - S' \Longrightarrow j\ a \in T - T'$
$\bigwedge b.\ b \in T - T' \Longrightarrow j\ (i\ b) = b$
$\bigwedge b.\ b \in T - T' \Longrightarrow i\ b \in S - S'$
assumes *nn*:
$\bigwedge a.\ a \in S' \Longrightarrow g\ a = z$
$\bigwedge b.\ b \in T' \Longrightarrow h\ b = z$
assumes *eq*:
$\bigwedge a.\ a \in S \Longrightarrow h\ (j\ a) = g\ a$
shows $F\ g\ S = F\ h\ T$
proof –
have *bij*: *bij-betw* $j\ (S - (S' \cap S))\ (T - (T' \cap T))$
using *witness* **by** (*intro bij-betw-byWitness*[**where** f'=i]) *auto*
have *F-eq*: $F\ g\ S = F\ (\lambda x.\ h\ (j\ x))\ S$
by (*intro cong*) (*auto simp*: *eq*)
show *?thesis*
unfolding *F-eq* **using** *fin nn eq*
by (*intro reindex-bij-betw-not-neutral*[*OF - - bij*]) *auto*
qed

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

lemma *delta* [*simp*]:
assumes *fS*: *finite S*
shows $F\ (\lambda k.$ *if* $k = a$ *then* $b\ k$ *else* $\mathbf{1})\ S = ($*if* $a \in S$ *then* $b\ a * \mathbf{1}$ *else* $\mathbf{1})$

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* $a \in S$ *then* $b\ a * \mathbf{1}$ *else* **1**)
using *delta* [*OF fin*, *of a b*, *symmetric*] **by** (*auto intro*: *cong*)

lemma *If-cases*:
fixes P :: $'b \Rightarrow$ *bool* **and** $g\ h$ :: $'b \Rightarrow 'a$
assumes *fin*: *finite A*
shows F (λx. *if* $P\ x$ *then* $h\ x$ *else* $g\ x$) $A = F\ h\ (A \cap \{x.\ P\ x\}) * F\ g\ (A \cap -\{x.\ P\ x\})$
proof −
have *a*: $A = A \cap \{x.\ P\ x\} \cup A \cap -\{x.\ P\ x\}$ $(A \cap \{x.\ P\ x\}) \cap (A \cap -\{x.\ P\ x\}) = \{\}$
by *blast*+
from *fin* **have** *f*: *finite* $(A \cap \{x.\ P\ x\})$ *finite* $(A \cap -\{x.\ P\ x\})$ **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*) $(A \times B)$
apply (*rule sym*)
apply (*cases finite A*)
apply (*cases finite B*)
apply (*simp add*: *Sigma*)
apply (*cases* $A = \{\}$)
apply *simp*
apply *simp*
apply (*auto intro*: *infinite dest*: *finite-cartesian-productD2*)
apply (*cases* $B = \{\}$)
apply (*auto intro*: *infinite dest*: *finite-cartesian-productD1*)
done

lemma *inter-restrict*:
assumes *finite A*
shows $F\ g\ (A \cap B) = F$ (λx. *if* $x \in B$ *then* $g\ x$ *else* **1**) A
proof −
let *?g* = λx. *if* $x \in A \cap B$ *then* $g\ x$ *else* **1**
have $\forall i \in A - A \cap B$. (*if* $i \in A \cap B$ *then* $g\ i$ *else* **1**) = **1** **by** *simp*
moreover have $A \cap B \subseteq A$ **by** *blast*
ultimately have F *?g* $(A \cap B) = F$ *?g* A
using ‹*finite A*› **by** (*intro mono-neutral-left*) *auto*
then show *?thesis* **by** *simp*
qed

lemma *inter-filter*:
finite A $\Longrightarrow F\ g\ \{x \in A.\ P\ x\} = F$ (λx. *if* $P\ x$ *then* $g\ x$ *else* **1**) A

by (*simp add*: *inter-restrict* [*symmetric*, *of A* {*x*. *P x*} *g*, *simplified*
mem-Collect-eq] *Int-def*)

lemma *Union-comp*:
assumes ∀ *A* ∈ *B*. *finite A*
and ⋀*A1 A2 x*. *A1* ∈ *B* ⟹ *A2* ∈ *B* ⟹ *A1* ≠ *A2* ⟹ *x* ∈ *A1* ⟹ *x* ∈ *A2*
⟹ *g x* = **1**
shows *F g* (⋃*B*) = (*F* ∘ *F*) *g B*
using *assms*
proof (*induct B rule*: *infinite-finite-induct*)
case (*infinite A*)
then have ¬ *finite* (⋃*A*) **by** (*blast dest*: *finite-UnionD*)
with *infinite* **show** *?case* **by** *simp*
next
case *empty*
then show *?case* **by** *simp*
next
case (*insert A B*)
then have *finite A finite B finite* (⋃*B*) *A* ∉ *B*
and ∀ *x*∈*A* ∩ ⋃*B*. *g x* = **1**
and *H*: *F g* (⋃*B*) = (*F* ∘ *F*) *g B* **by** *auto*
then have *F g* (*A* ∪ ⋃*B*) = *F g A* ∗ *F g* (⋃*B*)
by (*simp add*: *union-inter-neutral*)
with ‹*finite B*› ‹*A* ∉ *B*› **show** *?case*
by (*simp add*: *H*)
qed

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

lemma *swap-restrict*:
finite A ⟹ *finite B* ⟹
F (λ*x*. *F* (*g x*) {*y*. *y* ∈ *B* ∧ *R x y*}) *A* = *F* (λ*y*. *F* (λ*x*. *g x y*) {*x*. *x* ∈ *A* ∧ *R*
x y}) *B*
by (*simp add*: *inter-filter*) (*rule swap*)

lemma *Plus*:
fixes *A* :: *'b set* **and** *B* :: *'c set*
assumes *fin*: *finite A finite B*
shows *F g* (*A* <+> *B*) = *F* (*g* ∘ *Inl*) *A* ∗ *F* (*g* ∘ *Inr*) *B*
proof −
have *A* <+> *B* = *Inl* ‘ *A* ∪ *Inr* ‘ *B* **by** *auto*
moreover from *fin* **have** *finite* (*Inl* ‘ *A*) *finite* (*Inr* ‘ *B*) **by** *auto*
moreover have *Inl* ‘ *A* ∩ *Inr* ‘ *B* = {} **by** *auto*
moreover have *inj-on Inl A inj-on Inr B* **by** (*auto intro*: *inj-onI*)
ultimately show *?thesis*
using *fin* **by** (*simp add*: *union-disjoint reindex*)

qed

lemma *same-carrier*:
  assumes *finite* C
  assumes *subset*: $A \subseteq C$ $B \subseteq C$
  assumes *trivial*: $\bigwedge a.\ a \in C - A \Longrightarrow g\ a = \mathbf{1}$ $\bigwedge b.\ b \in C - B \Longrightarrow h\ b = \mathbf{1}$
  shows $F\ g\ A = F\ h\ B \longleftrightarrow F\ g\ C = F\ h\ C$
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** $C = A \cup (C - A)$ **by** *auto*
  then have $F\ g\ C = F\ g\ (A \cup (C - A))$ **by** *simp*
  also have $\ldots = F\ g\ (A - (C - A)) * F\ g\ (C - A - A) * F\ g\ (A \cap (C - A))$
    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** $C = B \cup (C - B)$ **by** *auto*
  then have $F\ h\ C = F\ h\ (B \cup (C - B))$ **by** *simp*
  also have $\ldots = F\ h\ (B - (C - B)) * F\ h\ (C - B - B) * F\ h\ (B \cap (C - B))$
    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-carrierI*:
  assumes *finite* C
  assumes *subset*: $A \subseteq C$ $B \subseteq C$
  assumes *trivial*: $\bigwedge a.\ a \in C - A \Longrightarrow g\ a = \mathbf{1}$ $\bigwedge b.\ b \in C - B \Longrightarrow h\ b = \mathbf{1}$
  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

Instead of $\sum x \mid P.\ e$ we introduce the shorter $\sum x|P.\ e$.

no-notation *Sum* (‹$\sum$›)

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* (‹$\sum$›)
where $\sum$ ≡ *sum-0* (λ*x*. *x*)

end

context *comm-monoid-add*
begin

subclass *ab-semigroup-add-0*
by *unfold-locales simp-all*

end

Now: lots of fancy syntax. First, *sum-0* (λ*x*. *e*) *A* is written $\sum$*x*∈*A*. *e*.

no-syntax (*ASCII*)
-sum :: *pttrn* ⇒ *'a set* ⇒ *'b* ⇒ *'b*::*comm-monoid-add* (‹(‹*indent=3* *notation=*‹*binder SUM*››*SUM* (-/:-)./ -)› [*0*, *51*, *10*] *10*)
no-syntax
-sum :: *pttrn* ⇒ *'a set* ⇒ *'b* ⇒ *'b*::*comm-monoid-add* (‹(‹*indent=2* *notation=*‹*binder* $\sum$››$\sum$(-/∈-)./ -)› [*0*, *51*, *10*] *10*)

syntax (*ASCII*)
-sum0 :: *pttrn* ⇒ *'a set* ⇒ *'b* ⇒ *'b*::*comm-monoid-add* (‹(‹*indent=3* *notation=*‹*binder SUM*››*SUM* (-/:-)./ -)› [*0*, *51*, *10*] *10*)
syntax
-sum0 :: *pttrn* ⇒ *'a set* ⇒ *'b* ⇒ *'b*::*comm-monoid-add* (‹(‹*indent=2* *notation=*‹*binder* $\sum$››$\sum$(-/∈-)./ -)› [*0*, *51*, *10*] *10*)
syntax-consts
-sum0 ⇌ *sum-0*
translations — Beware of argument permutation!
$\sum$*i*∈*A*. *b* ⇌ *CONST sum-0* (λ*i*. *b*) *A*

Instead of $\sum$*x*∈{*x*. *P*}. *e* we introduce the shorter $\sum$*x*|*P*. *e*.

no-syntax (*ASCII*)
-qsum :: *pttrn* ⇒ *bool* ⇒ *'a* ⇒ *'a* (‹(‹*indent=3* *notation=*‹*binder SUM Collect*››*SUM* - |/ -./ -)› [*0*, *0*, *10*] *10*)
no-syntax
-qsum :: *pttrn* ⇒ *bool* ⇒ *'a* ⇒ *'a* (‹(‹*indent=2* *notation=*‹*binder* $\sum$ *Collect*››$\sum$ - | (-)./ -)› [*0*, *0*, *10*] *10*)

syntax (*ASCII*)
-qsum0 :: *pttrn* ⇒ *bool* ⇒ *'a* ⇒ *'a* (‹(‹*indent=3* *notation=*‹*binder SUM Collect*››*SUM* - |/ -./ -)› [*0*, *0*, *10*] *10*)
syntax
-qsum0 :: *pttrn* ⇒ *bool* ⇒ *'a* ⇒ *'a* (‹(‹*indent=2* *notation=*‹*binder* $\sum$ *Collect*››$\sum$ - | (-)./ -)› [*0*, *0*, *10*] *10*)
syntax-consts
-qsum0 ⇌ *sum-0*

translations
  $\sum x|P.\ t$ => *CONST sum-0* $(\lambda x.\ t)\ \{x.\ P\}$
print-translation ‹
  [(***const-syntax*** ‹*sum-0*›, *K* (*Collect-binder-tr′* ***syntax-const*** ‹*-qsum*›))]
›

lemma (**in** *ab-semigroup-add-0*) *sum-image-gen-0*:
  assumes *fin*: *finite S*
  shows *sum-0 g S* = *sum-0* (λy. *sum-0 g* $\{x.\ x \in S \wedge f\ x = y\}$) (*f* ‘ *S*)
proof −
  have $\{y.\ y \in f`S \wedge f\ x = y\} = \{f\ x\}$ **if** $x \in S$ **for** *x*
    using *that* **by** *auto*
  then have *sum-0 g S* = *sum-0* (λx. *sum-0* (λy. *g x*) $\{y.\ y \in f`S \wedge f\ x = y\}$) *S*
    by *simp*
  also have . . . = *sum-0* (λy. *sum-0 g* $\{x.\ x \in S \wedge f\ x = y\}$) (*f* ‘ *S*)
    by (*rule sum-0.swap-restrict* [*OF fin finite-imageI* [*OF fin*]])
  finally show *?thesis* **.**
qed

2.2.1 Properties in more restricted classes of structures

lemma *sum-Un2*:
  assumes *finite* $(A \cup B)$
  shows *sum-0 f* $(A \cup B)$ = *sum-0 f* $(A - B)$ + *sum-0 f* $(B - A)$ + *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** *?thesis*
    by *simp* (*subst sum-0.union-disjoint*, *auto*)+
qed

class *ordered-ab-semigroup-add-0* = *ab-semigroup-add-0* +
ordered-ab-semigroup-add
begin

lemma *add-nonneg-nonneg* [*simp*]: $0 \le a \Longrightarrow 0 \le b \Longrightarrow 0 \le a + b$
  using *add-mono*[*of 0 a 0 b*] **by** *simp*

lemma *add-nonpos-nonpos*: $a \le 0 \Longrightarrow b \le 0 \Longrightarrow a + b \le 0$
  using *add-mono*[*of a 0 b 0*] **by** *simp*

end

lemma (**in** *ordered-ab-semigroup-add-0*) *sum-mono*:
  $(\bigwedge i.\ i \in K \Longrightarrow f\ i \le g\ i) \Longrightarrow (\sum i \in K.\ f\ i) \le (\sum i \in K.\ g\ i)$
  by (*induct K rule*: *infinite-finite-induct*) (*use add-mono* **in** *auto*)

lemma (**in** *ordered-ab-semigroup-add-0*) *sum-mono-00*:

```
  (⋀i. i∈K ⟹ f i + 0 ≤ g i + 0) ⟹ (∑ i∈K. f i) ≤ (∑ i∈K. g i)
proof (induct K rule: infinite-finite-induct)
  case (infinite A)
  then show ?case by simp
next
  case empty
  then show ?case 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 ∉ F
    assume a3: (⋀i. i ∈ F ⟹ f i + 0 ≤ g i + 0) ⟹ sum-0 f F ≤ sum-0 g F
    assume a4: ⋀i. i ∈ insert x F ⟹ f i + 0 ≤ g i + 0
    obtain bb :: 'b where
      f5: bb ∈ F ∧ ¬ f bb + 0 ≤ g bb + 0 ∨ sum-0 f F ≤ sum-0 g F
      using a3 by blast
    have ∀ b. x ≠ b ∨ f b + 0 ≤ g b + 0
      using a4 by simp
    then have ∀ a aa. f x + 0 + a ≤ g x + 0 + aa ∨ ¬ a ≤ aa
      using add-mono by blast
    then show sum-0 f (insert x F) ≤ 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
qed

lemma (in ordered-ab-semigroup-add-0) sum-mono-0:
  (⋀i. i∈K ⟹ f i + 0 ≤ g i) ⟹ (∑ i∈K. f i) ≤ (∑ i∈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: (⋀x. x ∈ A ⟹ 0 ≤ f x) ⟹ 0 ≤ 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 ≤ f x + sum-0 f F by (blast intro: add-mono)
  with insert show ?case by simp
qed
```

```
lemma sum-nonpos: (⋀x. x ∈ A ⟹ f x ≤ 0) ⟹ sum-0 f A ≤ 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 ≤ 0 + 0 by (blast intro: add-mono)
  with insert show ?case by simp
qed

lemma sum-mono2:
  assumes fin: finite B
    and sub: A ⊆ B
    and nn: ⋀b. b ∈ B−A ⟹ 0 ≤ f b
  shows sum-0 f A ≤ sum-0 f B
proof −
  have sum-0 f A ≤ 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 . . . = sum-0 f (A ∪ (B−A))
    by (simp add: sum-0.union-disjoint del: Un-Diff-cancel)
  also from sub have A ∪ (B−A) = B by blast
  finally show ?thesis .
qed

lemma sum-le-included:
  assumes finite s finite t
  and ∀ y∈t. 0 ≤ g y (∀ x∈s. ∃ y∈t. i y = x ∧ f x ≤ g y)
  shows sum-0 f s ≤ sum-0 g t
proof −
  have sum-0 f s ≤ sum-0 (λy. sum-0 g {x. x∈t ∧ i x = y}) s
  proof (rule sum-mono-0)
    fix y
    assume y ∈ s
    with assms obtain z where z: z ∈ t y = i z f y ≤ g z by auto
    hence f y + 0 ≤ 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-insertI)
    also have ... ≤ sum-0 g {x ∈ t. i x = y}
      apply (rule sum-mono2)
      using assms z by simp-all
    finally show f y + 0 ≤ sum-0 g {x ∈ t. i x = y} .
  qed
  also have . . . ≤ sum-0 (λy. sum-0 g {x. x∈t ∧ i x = y}) (i ‘ t)
    using assms(2−4) by (auto intro!: sum-mono2 sum-nonneg)
  also have . . . ≤ sum-0 g t
```

```
    using assms by (auto simp: sum-image-gen-0[symmetric])
  finally show ?thesis .
qed

end

lemma sum-comp-morphism:
  h 0 = 0 ⟹ (⋀x y. h (x + y) = h x + h y) ⟹ sum-0 (h ∘ g) A = h (sum-0 g
A)
  by (induct A rule: infinite-finite-induct) simp-all

lemma sum-cong-Suc:
  assumes 0 ∉ A ⋀x. Suc x ∈ A ⟹ f (Suc x) = g (Suc x)
  shows sum-0 f A = sum-0 g A
proof (rule sum-0.cong)
  fix x
  assume x ∈ 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

imports Stone-Kleene-Relation-Algebras.Kleene-Relation-Algebras

begin
```

context *sup*
begin

no-notation
sup (**infixl** ‹+› *65*)

end

context *plus*
begin

notation
plus (**infixl** ‹+› *65*)

end

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

class *sum* =
fixes *sum* :: $'a \Rightarrow 'a$

class *s-algebra* = *stone-relation-algebra* + *plus* + *sum* +
assumes *sum-plus-right-isotone*: $x \neq bot \wedge sum\ x \leq sum\ y \longrightarrow sum\ z + sum\ x \leq sum\ z + sum\ y$
assumes *sum-bot*: $sum\ x + sum\ bot = sum\ x$
assumes *sum-plus*: $sum\ x + sum\ y = sum\ (x \sqcup y) + sum\ (x \sqcap y)$
assumes *sum-conv*: $sum\ (x^T) = sum\ x$
begin

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

lemma *sum-symmetric*:
  assumes $y = y^T$
    shows $sum\ (x^T \sqcap y) = sum\ (x \sqcap y)$
  by (*metis assms sum-conv conv-dist-inf*)

lemma *sum-commute*:
  $sum\ x + sum\ y = sum\ y + sum\ x$
  by (*metis inf-commute sum-plus sup-commute*)

end

We next introduce the operation *minarc*. Axiom *minarc-below* expresses that the result of *minarc* is contained in the graph ignoring the weights. Axiom *minarc-arc* states that the result of *minarc* is a single unweighted edge if the graph is not empty. Axiom *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 *minarc*, where weights of edges between different nodes are compared by applying the operation *sum* to single-edge graphs. Axiom *sum-linear* requires that aggregated weights are linearly ordered, which is necessary for both Prim's and Kruskal's minimum spanning tree algorithms. Axiom *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 *minarc* =
  fixes $minarc :: 'a \Rightarrow 'a$

class *m-algebra* = *s-algebra* + *minarc* +
  assumes *minarc-below*: $minarc\ x \le --x$
  assumes *minarc-arc*: $x \ne bot \longrightarrow arc\ (minarc\ x)$
  assumes *minarc-min*: $arc\ y \wedge y \sqcap x \ne bot \longrightarrow sum\ (minarc\ x \sqcap x) \le sum\ (y$
$\sqcap\ x)$
  assumes *sum-linear*: $sum\ x \le sum\ y \vee sum\ y \le sum\ x$
  assumes *finite-regular*: $finite\ \{\ x\ .\ regular\ x\ \}$
begin

Axioms *minarc-below* and *minarc-arc* suffice to derive the Tarski rule in Stone relation algebras.

subclass *stone-relation-algebra-tarski*
proof *unfold-locales*
  fix x
  let $?a = minarc\ x$
  assume *1*: $regular\ x$
  assume $x \ne bot$
  hence $arc\ ?a$
    by (*simp add*: *minarc-arc*)
  hence $top = top * ?a * top$
    by (*simp add*: *comp-associative*)
  also have ... $\le top * --x * top$

by (*simp add*: *minarc-below mult-isotone*)
finally show $top * x * top = top$
using *1 order.antisym* **by** *simp*
qed

lemma *minarc-bot*:
minarc bot = bot
by (*metis bot-unique minarc-below regular-closed-bot*)

lemma *minarc-bot-iff*:
minarc x = bot $\longleftrightarrow$ *x = bot*
using *covector-bot-closed inf-bot-right minarc-arc vector-bot-closed minarc-bot*
by *fastforce*

lemma *minarc-meet-bot*:
assumes *minarc x* $\sqcap$ *x = bot*
shows *minarc x = 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 $\sqcap$ *x = bot* $\longleftrightarrow$ *minarc x = bot*
using *comp-inf.semiring.mult-not-zero minarc-meet-bot* **by** *blast*

lemma *minarc-meet-bot-iff*:
minarc x $\sqcap$ *x = bot* $\longleftrightarrow$ *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* $\sqcap$ *y*) *y*
using *inf-assoc minarc-regular selection-closed-id* **by** *auto*

lemma *minarc-below-regular*:

regular x ⟹ *minarc x* ≤ *x*
by (*metis minarc-below*)

end

class *m-kleene-algebra* = *m-algebra* + *stone-kleene-relation-algebra*

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*

imports *Stone-Kleene-Relation-Algebras.Matrix-Kleene-Algebras*
Aggregation-Algebras Semigroups-Big

begin

no-notation *inf* (**infixl** ‹⊓› *70*)
unbundle *no uminus-syntax*

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 $x \neq bot$ is necessary because the introduction of new edges can decrease the aggregated value. Axiom *add-bot* expresses that aggregation is zero-sum-free.

class *aggregation-order* = *order-bot* + *ab-semigroup-add* +
  assumes *add-right-isotone*: $x \neq bot \wedge x + bot \leq y + bot \longrightarrow x + z \leq y + z$
  assumes *add-add-bot* [*simp*]: $x + y + bot = x + y$
  assumes *add-bot*: $x + y = bot \longrightarrow x = bot$
begin

abbreviation *zero* $\equiv$ *bot* + *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*:
  $x + bot + bot + bot = x + bot$
  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.

syntax
  -sum-ab-semigroup-add-0 :: *idt* ⇒ *'a::bounded-semilattice-sup-bot* ⇒ *'a* (‹($\sum$-
-)› [*0*,*10*] *10*)

syntax-consts
  -sum-ab-semigroup-add-0 == *ab-semigroup-add-0.sum-0*

translations
  $\sum_x$ *t* => *XCONST ab-semigroup-add-0.sum-0 XCONST plus* (*XCONST plus XCONST bot XCONST bot*) (λx . *t*) { *x* . *CONST True* }

The following are basic properties of such sums.

lemma *agg-sum-bot*:
  ($\sum_k$ *bot::'a::aggregation-order*) = *bot* + *bot*
proof (*induct rule*: *infinite-finite-induct*)
  case (*infinite A*)
  thus *?case*

    by *simp*
next
  case *empty*
  thus *?case*
    by *simp*
next
  case (*insert x F*)
  thus *?case*
    by (*metis add.commute add-add-bot aggregation.sum-0.insert*)
qed

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

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

lemma *agg-sum-not-bot*:
  fixes f :: $('a::finite,'b::aggregation\text{-}order)$ *square*
  assumes $f\ (i,j) \neq bot$
    shows $(\sum_k \sum_l f\ (k,l)) \neq bot$
  by (*metis assms agg-sum-not-bot-1*)

lemma *agg-sum-distrib*:
  fixes $f\ g$ :: $'a \Rightarrow 'b::aggregation\text{-}order$
  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 $f\ g$ :: $('a,'b::aggregation\text{-}order)$ *square*
  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 ... $= (\sum_k \sum_l f\ (k,l)) + (\sum_k \sum_l g\ (k,l))$
    by (*metis* (*no-types*) *aggregation.sum-0.distrib*)
  finally show *?thesis*
    .
qed

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

lemma *agg-sum-add-bot-2*:
  fixes *f* :: *'a* ⇒ *'b*::*aggregation-order*
  shows $(\sum_k f\,k + bot) = (\sum_k f\,k)$
proof −
  have $(\sum_k f\,k + bot) = (\sum_k f\,k) + (\sum_k{::}'a\ bot{::}'b)$
    using *agg-sum-distrib* **by** *simp*
  also have ... $= (\sum_k f\,k) + (bot + bot)$
    by (*metis agg-sum-bot*)
  also have ... $= (\sum_k f\,k)$
    by *simp*
  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 $(\sum_l$ *if l = j then f l else zero*$) = f\,j + bot$
  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 $(\sum_l$ *if l = j then f l else bot*$) = f\,j + bot$
proof −
  let *?f* = (λ*l* . *if l = j then f l else bot*)
  let *?S* = {*l*::*'a* . *True*}
  show *?thesis*
  proof (*cases j* ∈ *?S*)
    case *False*
    thus *?thesis* **by** *simp*
  next
    case *True*
    let *?A* = *?S* − {*j*}
    let *?B* = {*j*}
    from *True* **have** *eq*: *?S* = *?A* ∪ *?B*
      by *blast*
    have *dj*: *?A* ∩ *?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* (λl . *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 ... = *zero* + (*f j* + *zero*)
   by *simp*
  also have ... = *f j* + *bot*
   by (*metis add.commute add.left-commute add-add-bot*)
  finally show *?thesis*
   .
 qed
qed

lemma *agg-delta-2*:
 fixes *f* :: (*'a::finite,'b::aggregation-order*) *square*
 shows ($\sum_k \sum_l$ *if* $k = i \wedge l = j$ *then f* (*k*,*l*) *else bot*) = *f* (*i*,*j*) + *bot*
proof −
 have $\forall k$. ($\sum_l$ *if* $k = i \wedge l = j$ *then f* (*k*,*l*) *else bot*) = (*if* $k = i$ *then f* (*k*,*j*) +
bot else zero)
 proof
  fix *k*
  have ($\sum_l$ *if* $k = i \wedge l = j$ *then f* (*k*,*l*) *else bot*) = ($\sum_l$ *if* $l = j$ *then if* $k = i$
then f (*k*,*l*) *else bot else bot*)
   by *meson*
  also have ... = (*if* $k = i$ *then f* (*k*,*j*) *else bot*) + *bot*
   by (*rule agg-delta-1*)
  finally show ($\sum_l$ *if* $k = i \wedge l = j$ *then f* (*k*,*l*) *else bot*) = (*if* $k = i$ *then f*
(*k*,*j*) + *bot else zero*)
   by *simp*
 qed
 hence ($\sum_k \sum_l$ *if* $k = i \wedge l = j$ *then f* (*k*,*l*) *else bot*) = ($\sum_k$ *if* $k = i$ *then f*
(*k*,*j*) + *bot else zero*)
  using *aggregation.sum-0.cong* **by** *auto*
 also have ... = *f* (*i*,*j*) + *bot*
  apply (*subst agg-delta*)
  by *simp*
 finally show *?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* :: ('a,'b::{plus,bot}) square (‹mzero›) **where** *mzero* = (λe . *bot* + *bot*)

definition *plus-matrix* :: ('a,'b::plus) square ⇒ ('a,'b) square ⇒ ('a,'b) square (**infixl** ‹$\oplus_M$› 65) **where** *plus-matrix f g* = (λe . *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* $\sum_k \sum_l$ *f (k,l) else bot + bot*)

Basic properties of these operations are given in the following.

lemma *bot-plus-bot*:
mbot $\oplus_M$ *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* $\sum_{(k::'a)} \sum_{(l::'a)}$ *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 ... = *mzero* (i,j)
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* $\oplus_M$ *mbot* = sum_M *f*
proof (*rule ext*, *rule prod-cases*)
let *?h* = *hd enum-class.enum*
fix *i j*
have (sum_M *f* $\oplus_M$ *mbot*) (i,j) = (*if i = ?h* ∧ *j = i then* ($\sum_k \sum_l$ *f (k,l)*) + *bot else zero + bot*)
by (*simp add*: *plus-matrix-def bot-matrix-def sum-matrix-def*)
also have ... = (*if i = ?h* ∧ *j = i then* $\sum_k \sum_l$ *f (k,l) else zero*)
by (*metis* (*no-types*, *lifting*) *add-add-bot aggregation.sum-0.F-one*)
also have ... = (sum_M *f*) (i,j)
by (*simp add*: *sum-matrix-def*)
finally show (sum_M *f* $\oplus_M$ *mbot*) (i,j) = (sum_M *f*) (i,j)
by *simp*
qed

lemma *sum-plus-zero*:
  fixes f :: ('a::enum,'b::aggregation-order) square
  shows $sum_M\ f \oplus_M\ mzero = sum_M\ f$
  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 $\forall\ i\ j\ .\ f\ (i,j) = bot$
    shows $f = mbot$
  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* :: ('a,'b::{plus,bot}) square ⇒ ('a,'b) square (‹$sum2_M$ -› [80] 80) **where** *sum-matrix-2* $f = (\lambda e\ .\ \sum_k \sum_l f\ (k,l))$

lemma *sum-bot-2*:
  $sum2_M$ (*mbot* :: ('a,'b::aggregation-order) square) = *mzero*
proof
  fix e
  have ($sum2_M$ *mbot* :: ('a,'b) square) $e = (\sum_{(k::'a)} \sum_{(l::'a)} bot)$
    by (*unfold sum-matrix-2-def bot-matrix-def*) *simp*
  also have ... = *bot* + *bot*
    using *agg-sum-bot aggregation.sum-0.neutral* **by** *fastforce*
  also have ... = *mzero* e
    by (*simp add*: *zero-matrix-def*)
  finally show ($sum2_M$ *mbot* :: ('a,'b) square) e = *mzero* e
    .
qed

lemma *sum-plus-bot-2*:
  fixes f :: ('a,'b::aggregation-order) square
  shows $sum2_M\ f \oplus_M\ mbot = sum2_M\ f$
proof
  fix e
  have ($sum2_M\ f \oplus_M\ mbot$) $e = (\sum_k \sum_l f\ (k,l)) + bot$
    by (*simp add*: *plus-matrix-def bot-matrix-def sum-matrix-2-def*)
  also have ... = $(\sum_k \sum_l f\ (k,l))$
    by (*metis* (*no-types*, *lifting*) *add-add-bot aggregation.sum-0.F-one*)
  also have ... = ($sum2_M\ f$) e
    by (*simp add*: *sum-matrix-2-def*)
  finally show ($sum2_M\ f \oplus_M\ mbot$) e = ($sum2_M\ f$) e
    by *simp*
qed

lemma *sum-plus-zero-2*:
fixes f :: ('*a*,'*b*::*aggregation-order*) *square*
shows $sum2_M\ f \oplus_M\ mzero = sum2_M\ f$
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 *add-lattice* implements the inclusion-exclusion principle at the level of edge weights.

class *aggregation-lattice* = *bounded-distrib-lattice* + *dense-lattice* + *aggregation-order* +
assumes *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 implementation of pseudocomplement.

class *aggregation-algebra* = *aggregation-lattice* + *uminus* + *one* + *times* + *conv* +
assumes *uminus-def* [*simp*]: $-x = ($*if* $x = bot$ *then top else bot*$)$
assumes *one-def* [*simp*]: *1* = *top*
assumes *times-def* [*simp*]: $x * y = x \sqcap y$
assumes *conv-def* [*simp*]: $x^T = x$
begin

subclass *stone-algebra*
apply *unfold-locales*
using *bot-meet-irreducible bot-unique* **by** *auto*

subclass *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 *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 $f \neq mbot \wedge sum_M\ f \preceq sum_M\ g \longrightarrow h \oplus_M\ sum_M\ f \preceq h \oplus_M\ sum_M\ g$

```
  proof
    let ?h = hd enum-class.enum
    assume 1: f ≠ mbot ∧ sum_M f ≼ sum_M g
    hence ∃ k l . f (k,l) ≠ bot
      by (meson agg-matrix-bot)
    hence 2: (∑_k ∑_l f (k,l)) ≠ bot
      using agg-sum-not-bot by blast
    have (∑_k ∑_l f (k,l)) = (sum_M f) (?h,?h)
      by (simp add: sum-matrix-def)
    also have ... ≤ (sum_M g) (?h,?h)
      using 1 by (simp add: less-eq-matrix-def)
    also have ... = (∑_k ∑_l g (k,l))
      by (simp add: sum-matrix-def)
    finally have (∑_k ∑_l f (k,l)) ≤ (∑_k ∑_l g (k,l))
      by simp
    hence 3: (∑_k ∑_l f (k,l)) + bot ≤ (∑_k ∑_l g (k,l)) + bot
      by (metis (no-types, lifting) add-add-bot aggregation.sum-0.F-one)
    show h ⊕_M sum_M f ≼ h ⊕_M sum_M g
    proof (unfold less-eq-matrix-def, rule allI, rule prod-cases, unfold
plus-matrix-def)
      fix i j
      have 4: h (i,j) + (∑_k ∑_l f (k,l)) ≤ h (i,j) + (∑_k ∑_l g (k,l))
        using 2 3 by (metis (no-types, lifting) add-right-isotone add.commute)
      have h (i,j) + (sum_M f) (i,j) = h (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 h (i,j) + (∑_k ∑_l f (k,l)) else h
(i,j) + zero)
        by simp
      also have ... ≤ (if i = ?h ∧ j = i then h (i,j) + (∑_k ∑_l g (k,l)) else h
(i,j) + zero)
        using 4 order.eq-iff by auto
      also have ... = h (i,j) + (if i = ?h ∧ j = i then ∑_k ∑_l g (k,l) else zero)
        by simp
      finally show h (i,j) + (sum_M f) (i,j) ≤ h (i,j) + (sum_M g) (i,j)
        by (simp add: sum-matrix-def)
    qed
  qed
next
  fix f :: ('a,'b) square
  show sum_M f ⊕_M sum_M mbot = sum_M f
    by (simp add: sum-bot sum-plus-zero)
next
  fix f g :: ('a,'b) square
  show sum_M f ⊕_M sum_M g = sum_M (f ⊕ g) ⊕_M sum_M (f ⊗ g)
  proof (rule ext, rule prod-cases)
    fix i j
    let ?h = hd enum-class.enum
    have (sum_M f ⊕_M sum_M g) (i,j) = (sum_M f) (i,j) + (sum_M g) (i,j)
```

by (*simp add*: *plus-matrix-def*)
also have ... = (*if* i = *?h* ∧ j = i *then* $\sum_k \sum_l f\ (k,l)$ *else zero*) + (*if* i = *?h* ∧ j = i *then* $\sum_k \sum_l g\ (k,l)$ *else zero*)
by (*simp add*: *sum-matrix-def*)
also have ... = (*if* i = *?h* ∧ j = i *then* $(\sum_k \sum_l f\ (k,l)) + (\sum_k \sum_l g\ (k,l))$ *else zero*)
by *simp*
also have ... = (*if* i = *?h* ∧ j = i *then* $\sum_k \sum_l f\ (k,l) + g\ (k,l)$ *else zero*)
using *agg-sum-distrib-2* **by** (*metis* (*no-types*))
also have ... = (*if* i = *?h* ∧ j = i *then* $\sum_k \sum_l (f\ (k,l) \sqcup g\ (k,l)) + (f\ (k,l) \sqcap g\ (k,l))$ *else zero*)
using *add-lattice aggregation.sum-0.cong* **by** (*metis* (*no-types*, *lifting*))
also have ... = (*if* i = *?h* ∧ j = i *then* $\sum_k \sum_l (f \oplus g)\ (k,l) + (f \otimes g)\ (k,l)$ *else zero*)
by (*simp add*: *sup-matrix-def inf-matrix-def*)
also have ... = (*if* i = *?h* ∧ j = i *then* $(\sum_k \sum_l (f \oplus g)\ (k,l)) + (\sum_k \sum_l (f \otimes g)\ (k,l))$ *else zero*)
using *agg-sum-distrib-2* **by** (*metis* (*no-types*))
also have ... = (*if* i = *?h* ∧ j = i *then* $\sum_k \sum_l (f \oplus g)\ (k,l)$ *else zero*) + (*if* i = *?h* ∧ j = i *then* $\sum_k \sum_l (f \otimes g)\ (k,l)$ *else zero*)
by *simp*
also have ... = $(sum_M\ (f \oplus g))\ (i,j) + (sum_M\ (f \otimes g))\ (i,j)$
by (*simp add*: *sum-matrix-def*)
also have ... = $(sum_M\ (f \oplus g) \oplus_M sum_M\ (f \otimes g))\ (i,j)$
by (*simp add*: *plus-matrix-def*)
finally show $(sum_M\ f \oplus_M sum_M\ g)\ (i,j) = (sum_M\ (f \oplus g) \oplus_M sum_M\ (f \otimes g))\ (i,j)$
.
qed
next
fix f :: ('a,'b) *square*
show $sum_M\ (f^t) = sum_M\ f$
proof (*rule ext*, *rule prod-cases*)
fix $i\ j$
let *?h* = *hd enum-class.enum*
have $(sum_M\ (f^t))\ (i,j)$ = (*if* i = *?h* ∧ j = i *then* $\sum_k \sum_l (f^t)\ (k,l)$ *else zero*)
by (*simp add*: *sum-matrix-def*)
also have ... = (*if* i = *?h* ∧ j = i *then* $\sum_k \sum_l (f\ (l,k))^T$ *else zero*)
by (*simp add*: *conv-matrix-def*)
also have ... = (*if* i = *?h* ∧ j = i *then* $\sum_k \sum_l f\ (l,k)$ *else zero*)
by *simp*
also have ... = (*if* i = *?h* ∧ j = i *then* $\sum_l \sum_k f\ (l,k)$ *else zero*)
by (*metis agg-sum-commute*)
also have ... = $(sum_M\ f)\ (i,j)$
by (*simp add*: *sum-matrix-def*)
finally show $(sum_M\ (f^t))\ (i,j) = (sum_M\ f)\ (i,j)$
.
qed
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
  fix $f\ g\ h$ :: (*'a*,*'b*) *square*
  show $f \neq mbot \wedge sum2_M\ f \preceq sum2_M\ g \longrightarrow h \oplus_M sum2_M\ f \preceq h \oplus_M sum2_M\ g$
  proof
    assume *1*: $f \neq mbot \wedge sum2_M\ f \preceq sum2_M\ g$
    hence $\exists k\ l\ .\ f\ (k,l) \neq bot$
      by (*meson agg-matrix-bot*)
    hence *2*: $(\sum_k \sum_l f\ (k,l)) \neq bot$
      using *agg-sum-not-bot* **by** *blast*
    obtain *c* :: *'a* **where** *True*
      by *simp*
    have $(\sum_k \sum_l f\ (k,l)) = (sum2_M\ f)\ (c,c)$
      by (*simp add*: *sum-matrix-2-def*)
    also have ... $\leq (sum2_M\ g)\ (c,c)$
      using *1* **by** (*simp add*: *less-eq-matrix-def*)
    also have ... $= (\sum_k \sum_l g\ (k,l))$
      by (*simp add*: *sum-matrix-2-def*)
    finally have $(\sum_k \sum_l f\ (k,l)) \leq (\sum_k \sum_l g\ (k,l))$
      by *simp*
    hence *3*: $(\sum_k \sum_l f\ (k,l)) + bot \leq (\sum_k \sum_l g\ (k,l)) + bot$
      by (*metis* (*no-types*, *lifting*) *add-add-bot aggregation.sum-0.F-one*)
    show $h \oplus_M sum2_M\ f \preceq h \oplus_M sum2_M\ g$
    proof (*unfold less-eq-matrix-def*, *rule allI*, *unfold plus-matrix-def*)
      fix *e*
      have $h\ e + (sum2_M\ f)\ e = h\ e + (\sum_k \sum_l f\ (k,l))$
        by (*simp add*: *sum-matrix-2-def*)
      also have ... $\leq h\ e + (\sum_k \sum_l g\ (k,l))$
        using *2 3* **by** (*metis* (*no-types*, *lifting*) *add-right-isotone add.commute*)
      finally show $h\ e + (sum2_M\ f)\ e \leq h\ e + (sum2_M\ g)\ e$
        by (*simp add*: *sum-matrix-2-def*)
    qed
  qed
next
  fix *f* :: (*'a*,*'b*) *square*
  show $sum2_M\ f \oplus_M sum2_M\ mbot = sum2_M\ f$
    by (*simp add*: *sum-bot-2 sum-plus-zero-2*)
next
  fix *f g* :: (*'a*,*'b*) *square*
  show $sum2_M\ f \oplus_M sum2_M\ g = sum2_M\ (f \oplus g) \oplus_M sum2_M\ (f \otimes g)$
  proof
    fix *e*

have $(sum2_M\ f \oplus_M sum2_M\ g)\ e = (sum2_M\ f)\ e + (sum2_M\ g)\ e$
by (*simp add*: *plus-matrix-def*)
also have ... $= (\sum_k \sum_l f\ (k,l)) + (\sum_k \sum_l g\ (k,l))$
by (*simp add*: *sum-matrix-2-def*)
also have ... $= (\sum_k \sum_l f\ (k,l) + g\ (k,l))$
using *agg-sum-distrib-2* **by** (*metis* (*no-types*))
also have ... $= (\sum_k \sum_l (f\ (k,l) \sqcup g\ (k,l)) + (f\ (k,l) \sqcap g\ (k,l)))$
using *add-lattice aggregation.sum-0.cong* **by** (*metis* (*no-types*, *lifting*))
also have ... $= (\sum_k \sum_l (f \oplus g)\ (k,l) + (f \otimes g)\ (k,l))$
by (*simp add*: *sup-matrix-def inf-matrix-def*)
also have ... $= (\sum_k \sum_l (f \oplus g)\ (k,l)) + (\sum_k \sum_l (f \otimes g)\ (k,l))$
using *agg-sum-distrib-2* **by** (*metis* (*no-types*))
also have ... $= (sum2_M\ (f \oplus g))\ e + (sum2_M\ (f \otimes g))\ e$
by (*simp add*: *sum-matrix-2-def*)
also have ... $= (sum2_M\ (f \oplus g) \oplus_M sum2_M\ (f \otimes g))\ e$
by (*simp add*: *plus-matrix-def*)
finally show $(sum2_M\ f \oplus_M sum2_M\ g)\ e = (sum2_M\ (f \oplus g) \oplus_M sum2_M\ (f \otimes g))\ e$
.
qed
next
fix f :: ('a,'b) *square*
show $sum2_M\ (f^t) = sum2_M\ f$
proof
fix e
have $(sum2_M\ (f^t))\ e = (\sum_k \sum_l (f^t)\ (k,l))$
by (*simp add*: *sum-matrix-2-def*)
also have ... $= (\sum_k \sum_l (f\ (l,k))^T)$
by (*simp add*: *conv-matrix-def*)
also have ... $= (\sum_k \sum_l f\ (l,k))$
by *simp*
also have ... $= (\sum_l \sum_k f\ (l,k))$
by (*metis agg-sum-commute*)
also have ... $= (sum2_M\ f)\ e$
by (*simp add*: *sum-matrix-2-def*)
finally show $(sum2_M\ (f^t))\ e = (sum2_M\ f)\ e$
.
qed
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#ys*) x = (*if* $x = y$ *then* 0 *else* 1 + *enum-pos' ys* x)

lemma *enum-pos′-inverse*:
List.member xs x $\Longrightarrow$ *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-less1 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* $\Longrightarrow$ *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 x* ≤ *enum-pos y*)
abbreviation *enum-pos-less* :: ′*a*::*enum* ⇒ ′*a* ⇒ *bool* **where** *enum-pos-less x y* ≡ (*enum-pos x* < *enum-pos y*)

sublocale *enum* < *enum-order*: *order* **where** *less-eq* = λ*x y* . *enum-pos-less-eq x y* **and** *less* = λ*x y* . *enum-pos x* < *enum-pos y*
apply *unfold-locales*
by *auto*

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 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-⊥ minimum element which is first in the lexicographic order. The result is returned as a regular matrix with ⊤ at that location and ⊥ 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* = (λ*e* . *if f e* ≠ *bot* ∧ (∀ *d*

. $(f\ d \neq bot \longrightarrow (f\ e + bot \leq f\ d + bot \wedge (enum\text{-}lex\text{-}less\ d\ e \longrightarrow f\ e + bot \neq f\ d + bot))))$ *then top else bot*)

lemma *minarc-at-most-one*:
fixes f :: ('a::enum,'b::{aggregation-order,top}) *square*
assumes $(minarc_M\ f)\ e \neq bot$
and $(minarc_M\ f)\ d \neq bot$
shows $e = d$
proof –
have *1*: $f\ e + bot \leq f\ d + bot$
by (*metis assms minarc-matrix-def*)
have $f\ d + bot \leq f\ e + bot$
by (*metis assms minarc-matrix-def*)
hence $f\ e + bot = f\ d + bot$
using *1* **by** *simp*
hence $\neg$ *enum-lex-less d e* $\wedge$ $\neg$ *enum-lex-less e d*
using *assms* **by** (*unfold minarc-matrix-def*) (*metis* (*lifting*))
thus *?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* $\wedge$ *le y z* $\longrightarrow$ *le x z*
and *total*: $\forall x\ y$. *le x y* $\vee$ *le y x*
shows *finite A* $\Longrightarrow A \neq \{\} \Longrightarrow \exists x . x \in A \wedge (\forall y . y \in A \longrightarrow$ *le x y*)
proof (*induct A rule*: *finite-ne-induct*)
case *singleton*
thus *?case*

```
    using total by auto
next
  case insert
  thus ?case
    by (metis insert-iff transitive total)
qed

lemma minarc-at-least-one:
  fixes f :: ('a::enum,'b::linear-aggregation-lattice) square
  assumes f ≠ mbot
    shows ∃ e . (minarc_M f) e = top
proof −
  let ?nbe = { (e,f e) | e . f e ≠ bot }
  have 1: finite ?nbe
    using finite-code finite-image-set by blast
  have 2: ?nbe ≠ {}
    using assms agg-matrix-bot by fastforce
  let ?le = λ(e::'a × 'a,fe::'b) (d::'a × 'a,fd) . fe + bot ≤ fd + bot
  have 3: ∀ x y z . ?le x y ∧ ?le y z ⟶ ?le x z
    by auto
  have 4: ∀ x y . ?le x y ∨ ?le y x
    by (simp add: linear)
  have ∃ x . x ∈ ?nbe ∧ (∀ y . y ∈ ?nbe ⟶ ?le x y)
    by (rule least-order, rule 3, rule 4, rule 1, rule 2)
  then obtain e fe where 5: (e,fe) ∈ ?nbe ∧ (∀ y . y ∈ ?nbe ⟶ ?le (e,fe) y)
    by auto
  let ?me = { e . f e ≠ bot ∧ f e + bot = fe + bot }
  have 6: finite ?me
    using finite-code finite-image-set by blast
  have 7: ?me ≠ {}
    using 5 by auto
  have 8: ∀ x y z . enum-lex-less-eq x y ∧ enum-lex-less-eq y z ⟶
enum-lex-less-eq x z
    by auto
  have 9: ∀ x y . enum-lex-less-eq x y ∨ enum-lex-less-eq y x
    by auto
  have ∃ x . x ∈ ?me ∧ (∀ y . y ∈ ?me ⟶ enum-lex-less-eq x y)
    by (rule least-order, rule 8, rule 9, rule 6, rule 7)
  then obtain m where 10: m ∈ ?me ∧ (∀ y . y ∈ ?me ⟶ enum-lex-less-eq m
y)
    by auto
  have 11: f m ≠ bot
    using 10 5 by auto
  have 12: ∀ d. f d ≠ bot ⟶ f m + bot ≤ f d + bot
    using 10 5 by simp
  have ∀ d. f d ≠ bot ∧ enum-lex-less d m ⟶ f m + bot ≠ f d + 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* + *times* + *conv* +
assumes *uminus-def-2* [*simp*]: $-x = (\textit{if } x = \textit{bot then top else bot})$
assumes *one-def-2* [*simp*]: $1 = \textit{top}$
assumes *times-def-2* [*simp*]: $x * y = x \sqcap y$
assumes *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 $x \longleftrightarrow x = \textit{bot} \vee x = \textit{top}$
by *simp*

sublocale *heyting*: *heyting-stone-algebra* **where** *implies* = $\lambda x\ y\ .\ \textit{if } x \leq y \textit{ then top else } y$
apply *unfold-locales*
apply (*simp add*: *order.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*: *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 $\textit{minarc}_M\ f \preceq \ominus\ominus f$
proof (*unfold less-eq-matrix-def*, *rule allI*)
fix e :: '*a* × '*a*
have $(\textit{minarc}_M\ f)\ e \leq (\textit{if } f\ e \neq \textit{bot then top else } --(f\ e))$
by (*simp add*: *minarc-matrix-def*)
also have ... $= --(f\ e)$
by *simp*
also have ... $= (\ominus\ominus f)\ e$

by (*simp add*: *uminus-matrix-def*)
finally show (*minarc*$_M$ *f*) *e* ≤ (⊖⊖*f*) *e*
.
qed
next
fix *f* :: (′*a*,′*b*) *square*
let *?at* = *bounded-distrib-allegory-signature.arc mone times-matrix less-eq-matrix mtop conv-matrix*
show *f* ≠ *mbot* ⟶ *?at* (*minarc*$_M$ *f*)
proof
assume *1*: *f* ≠ *mbot*
have *minarc*$_M$ *f* ⊙ *mtop* ⊙ (*minarc*$_M$ *f* ⊙ *mtop*)^t = *minarc*$_M$ *f* ⊙ *mtop* ⊙ (*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 ... ≼ *mone*
proof (*unfold less-eq-matrix-def*, *rule allI*, *rule prod-cases*)
fix *i j*
have (*minarc*$_M$ *f* ⊙ *mtop* ⊙ (*minarc*$_M$ *f*)^t) (*i*,*j*) = (⨆$_l$ (⨆$_k$ (*minarc*$_M$ *f*) (*i*,*k*) ∗ *mtop* (*k*,*l*)) ∗ ((*minarc*$_M$ *f*)^t) (*l*,*j*))
by (*simp add*: *times-matrix-def*)
also have ... = (⨆$_l$ (⨆$_k$ (*minarc*$_M$ *f*) (*i*,*k*) ∗ *top*) ∗ ((*minarc*$_M$ *f*) (*j*,*l*))^T)
by (*simp add*: *top-matrix-def conv-matrix-def*)
also have ... = (⨆$_l$ ⨆$_k$ (*minarc*$_M$ *f*) (*i*,*k*) ∗ *top* ∗ ((*minarc*$_M$ *f*) (*j*,*l*))^T)
by (*metis comp-right-dist-sum*)
also have ... = (⨆$_l$ ⨆$_k$ *if i* = *j* ∧ *l* = *k then* (*minarc*$_M$ *f*) (*i*,*k*) ∗ *top* ∗ ((*minarc*$_M$ *f*) (*j*,*l*))^T *else bot*)
apply (*rule sup-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* (⨆$_l$ ⨆$_k$ *if l* = *k then* (*minarc*$_M$ *f*) (*i*,*k*) ∗ *top* ∗ ((*minarc*$_M$ *f*) (*j*,*l*))^T *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* ⊙ *mtop* ⊙ (*minarc*$_M$ *f*)^t) (*i*,*j*) ≤ *mone* (*i*,*j*)
.
qed
finally have *2*: *minarc*$_M$ *f* ⊙ *mtop* ⊙ (*minarc*$_M$ *f* ⊙ *mtop*)^t ≼ *mone*
.
have *3*: *mtop* ⊙ (*minarc*$_M$ *f* ⊙ *mtop*) = *mtop*
proof (*rule ext*, *rule prod-cases*)
fix *i j*
from *minarc-at-least-one* **obtain** *ei ej* **where** (*minarc*$_M$ *f*) (*ei*,*ej*) = *top*
using *1* **by** *force*

hence *4*: $top * top \le (\bigsqcup_l (minarc_M\ f)\ (ei,l) * top)$
by (*metis comp-inf.ub-sum*)
have $top * (\bigsqcup_l (minarc_M\ f)\ (ei,l) * top) \le (\bigsqcup_k top * (\bigsqcup_l (minarc_M\ f)\ (k,l) * top))$
by (*rule comp-inf.ub-sum*)
hence $top \le (\bigsqcup_k top * (\bigsqcup_l (minarc_M\ f)\ (k,l) * top))$
using *4* **by** *auto*
also have ... $= (\bigsqcup_k mtop\ (i,k) * (\bigsqcup_l (minarc_M\ f)\ (k,l) * mtop\ (l,j)))$
by (*simp add*: *top-matrix-def*)
also have ... $= (mtop \odot (minarc_M\ f \odot mtop))\ (i,j)$
by (*simp add*: *times-matrix-def*)
finally show $(mtop \odot (minarc_M\ f \odot mtop))\ (i,j) = mtop\ (i,j)$
by (*simp add*: *eq-iff top-matrix-def*)
qed
have $(minarc_M\ f)^t \odot mtop \odot ((minarc_M\ f)^t \odot mtop)^t = (minarc_M\ f)^t \odot mtop \odot (minarc_M\ f)$
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.surjective-top-closed*)
also have ... $\preceq mone$
proof (*unfold less-eq-matrix-def*, *rule allI*, *rule prod-cases*)
fix *i j*
have $((minarc_M\ f)^t \odot mtop \odot minarc_M\ f)\ (i,j) = (\bigsqcup_l (\bigsqcup_k ((minarc_M\ f)^t)\ (i,k) * mtop\ (k,l)) * (minarc_M\ f)\ (l,j))$
by (*simp add*: *times-matrix-def*)
also have ... $= (\bigsqcup_l (\bigsqcup_k ((minarc_M\ f)\ (k,i))^T * top) * (minarc_M\ f)\ (l,j))$
by (*simp add*: *top-matrix-def conv-matrix-def*)
also have ... $= (\bigsqcup_l \bigsqcup_k ((minarc_M\ f)\ (k,i))^T * top * (minarc_M\ f)\ (l,j))$
by (*metis comp-right-dist-sum*)
also have ... $= (\bigsqcup_l \bigsqcup_k if\ i = j \wedge l = k\ then\ ((minarc_M\ f)\ (k,i))^T * top * (minarc_M\ f)\ (l,j)\ else\ bot)$
apply (*rule sup-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 ... $= (if\ i = j\ then\ (\bigsqcup_l \bigsqcup_k if\ l = k\ then\ ((minarc_M\ f)\ (k,i))^T * top * (minarc_M\ f)\ (l,j)\ else\ bot)\ else\ bot)$
by *auto*
also have ... $\le (if\ i = j\ then\ top\ else\ bot)$
by *simp*
also have ... $= mone\ (i,j)$
by (*simp add*: *one-matrix-def*)
finally show $((minarc_M\ f)^t \odot mtop \odot (minarc_M\ f))\ (i,j) \le mone\ (i,j)$
.
qed
finally have *5*: $(minarc_M\ f)^t \odot mtop \odot ((minarc_M\ f)^t \odot mtop)^t \preceq mone$
.

have $mtop \odot ((minarc_M\ f)^t \odot mtop) = mtop$
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 $?at\ g \wedge g \otimes f \neq mbot \longrightarrow sum_M\ (minarc_M\ f \otimes f) \preceq sum_M\ (g \otimes f)$
proof
assume *1*: $?at\ g \wedge g \otimes f \neq mbot$
hence *2*: $g = \ominus\ominus g$
using *matrix-stone-relation-algebra.arc-regular* **by** *blast*
show $sum_M\ (minarc_M\ f \otimes f) \preceq sum_M\ (g \otimes f)$
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*: $\forall\, k\ l\ .\ \neg(k = ei \wedge l = ej) \longrightarrow (minarc_M\ f)\ (k,l) = bot$
by (*metis* (*mono-tags*, *opaque-lifting*) *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 uminus-matrix-def uminus-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 ... $\leq (\bigsqcup_n g\ (di,n) * top) * (g^t)\ (l,k)$
by (*metis comp-inf.ub-sum comp-right-dist-sum*)
also have ... $\leq (\bigsqcup_m (\bigsqcup_n g\ (di,n) * top) * (g^t)\ (m,k))$
by (*metis comp-inf.ub-sum*)
also have ... $= (g \odot mtop \odot g^t)\ (di,k)$
by (*simp add*: *times-matrix-def top-matrix-def*)
also have ... $\leq mone\ (di,k)$
using *1* **by** (*metis matrix-stone-relation-algebra.arc-expanded less-eq-matrix-def*)
also have ... $= bot$
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 uminus-def uminus-matrix-def*)
qed
have $\forall\, k\ l\ .\ l \neq dj \longrightarrow g\ (k,l) = bot$
proof (*intro allI*, *rule impI*)
fix *k l*
assume *11*: $l \neq dj$
have $(g\ (k,l))^T * top = (g^t)\ (l,k) * top * g\ (di,dj)$
using *7* **by** (*simp add*: *comp-associative conv-matrix-def*)
also have ... $\leq (\bigsqcup_n\ (g^t)\ (l,n) * top) * g\ (di,dj)$
by (*metis comp-inf.ub-sum comp-right-dist-sum*)
also have ... $\leq (\bigsqcup_m\ (\bigsqcup_n\ (g^t)\ (l,n) * top) * g\ (m,dj))$
by (*metis comp-inf.ub-sum*)
also have ... $= (g^t \odot mtop \odot g)\ (l,dj)$
by (*simp add*: *times-matrix-def top-matrix-def*)
also have ... $\leq mone\ (l,dj)$
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 $g\ (k,l) \neq top$
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*: $\forall\, k\ l\ .\ \neg(k = di \wedge l = dj) \longrightarrow (g \otimes f)\ (k,l) = bot$
using *9* **by** (*metis inf-bot-left inf-matrix-def*)
have $(\sum_k \sum_l\ (minarc_M\ f \otimes f)\ (k,l)) = (\sum_k \sum_l\ if\ k = ei \wedge l = ej\ then\ (minarc_M\ f \otimes f)\ (k,l)\ else\ (minarc_M\ f \otimes f)\ (k,l))$
by *simp*
also have ... $= (\sum_k \sum_l\ if\ k = ei \wedge l = ej\ then\ (minarc_M\ f \otimes f)\ (k,l)\ else\ (minarc_M\ f)\ (k,l) \sqcap f\ (k,l))$
by (*unfold inf-matrix-def*) *simp*
also have ... $= (\sum_k \sum_l\ if\ k = ei \wedge l = ej\ then\ (minarc_M\ f \otimes f)\ (k,l)\ else\ bot)$
apply (*rule aggregation.sum-0.cong*)
apply *simp*
using *4* **by** (*metis inf-bot-left*)
also have ... $= (minarc_M\ f \otimes f)\ (ei,ej) + bot$
by (*unfold agg-delta-2*) *simp*
also have ... $= f\ (ei,ej) + bot$
using *3* **by** (*simp add*: *inf-matrix-def*)
also have ... $\leq (g \otimes f)\ (di,dj) + bot$

using *3 5 6 7 8* **by** (*metis minarc-matrix-def*)
also have ... = ($\sum_k \sum_l$ *if* $k = di \wedge l = dj$ *then* $(g \otimes f)$ (k,l) *else bot*)
by (*unfold agg-delta-2*) *simp*
also have ... = ($\sum_k \sum_l$ *if* $k = di \wedge l = dj$ *then* $(g \otimes f)$ (k,l) *else* $(g \otimes f)$ (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 (sum_M ($minarc_M$ $f \otimes f$)) $(i,j) \leq$ (sum_M $(g \otimes f)$) (i,j)
by (*simp add*: *sum-matrix-def*)
qed
qed
next
fix *f g* :: ($'a$,$'b$) *square*
let *?h* = *hd enum-class.enum*
show sum_M $f \preceq sum_M$ $g \vee sum_M$ $g \preceq sum_M$ f
proof (*cases* (sum_M f) (*?h*,*?h*) $\leq$ (sum_M g) (*?h*,*?h*))
case *1*: *True*
have sum_M $f \preceq sum_M$ g
apply (*unfold less-eq-matrix-def*, *rule allI*, *rule prod-cases*)
using *1* **by** (*unfold sum-matrix-def*) *auto*
thus *?thesis*
by *simp*
next
case *False*
hence *2*: (sum_M g) (*?h*,*?h*) $\leq$ (sum_M f) (*?h*,*?h*)
by (*simp add*: *linear*)
have sum_M $g \preceq sum_M$ f
apply (*unfold less-eq-matrix-def*, *rule allI*, *rule prod-cases*)
using *2* **by** (*unfold sum-matrix-def*) *auto*
thus *?thesis*
by *simp*
qed
next
have *finite* { *f* :: ($'a$,$'b$) *square* . ($\forall$ *e* . *regular* (*f e*)) }
by (*unfold regular-bot-top-2*, *rule finite-set-of-finite-funs-pred*) *auto*
thus *finite* { *f* :: ($'a$,$'b$) *square* . *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*: *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-2* **and**

minarc = *minarc-matrix*
proof
  fix *f* :: ('*a*,'*b*) *square*
  show $minarc_M$ *f* ≼ ⊖⊖*f*
    by (*simp add*: *agg-square-m-algebra.minarc-below*)
next
  fix *f* :: ('*a*,'*b*) *square*
  let *?at* = *bounded-distrib-allegory-signature.arc mone times-matrix less-eq-matrix mtop conv-matrix*
  show *f* ≠ *mbot* ⟶ *?at* ($minarc_M$ *f*)
    by (*simp add*: *agg-square-m-algebra.minarc-arc*)
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* ∧ *g* ⊗ *f* ≠ *mbot* ⟶ $sum2_M$ ($minarc_M$ *f* ⊗ *f*) ≼ $sum2_M$ (*g* ⊗ *f*)
  proof
    let *?h* = *hd enum-class.enum*
    assume *?at g* ∧ *g* ⊗ *f* ≠ *mbot*
    hence sum_M ($minarc_M$ *f* ⊗ *f*) ≼ sum_M (*g* ⊗ *f*)
      by (*simp add*: *agg-square-m-algebra.minarc-min*)
    hence (sum_M ($minarc_M$ *f* ⊗ *f*)) (*?h*,*?h*) ≤ (sum_M (*g* ⊗ *f*)) (*?h*,*?h*)
      by (*simp add*: *less-eq-matrix-def*)
    thus $sum2_M$ ($minarc_M$ *f* ⊗ *f*) ≼ $sum2_M$ (*g* ⊗ *f*)
      by (*simp add*: *sum-matrix-def sum-matrix-2-def less-eq-matrix-def*)
  qed
next
  fix *f g* :: ('*a*,'*b*) *square*
  let *?h* = *hd enum-class.enum*
  have sum_M *f* ≼ sum_M *g* ∨ sum_M *g* ≼ sum_M *f*
    by (*simp add*: *agg-square-m-algebra.sum-linear*)
  hence (sum_M *f*) (*?h*,*?h*) ≤ (sum_M *g*) (*?h*,*?h*) ∨ (sum_M *g*) (*?h*,*?h*) ≤ (sum_M *f*) (*?h*,*?h*)
    using *less-eq-matrix-def* **by** *auto*
  thus $sum2_M$ *f* ≼ $sum2_M$ *g* ∨ $sum2_M$ *g* ≼ $sum2_M$ *f*
    by (*simp add*: *sum-matrix-def sum-matrix-2-def less-eq-matrix-def*)
next
  show *finite* { *f* :: ('*a*,'*b*) *square* . *matrix-p-algebra.regular f* }
    by (*simp add*: *agg-square-m-algebra.finite-regular*)
qed

By defining the Kleene star as ⊤ aggregation lattices form a Kleene algebra.

class *aggregation-kleene-algebra* = *aggregation-algebra* + *star* +
  assumes *star-def* [*simp*]: $x^\star$ = *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*]: $x^\star$ = *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* **..**

class *linorder-stone-relation-algebra-plus-expansion* = *linorder-stone-relation-algebra-expansion* + *plus* +
assumes *plus-def*: *plus* = *sup*
begin

subclass *linear-aggregation-algebra*
apply *unfold-locales*
using *plus-def sup-monoid.add-assoc* **apply** *blast*
using *plus-def sup-monoid.add-commute* **apply** *blast*
using *comp-inf.semiring.add-mono plus-def* **apply** *auto*[*1*]
using *plus-def* **apply** *force*
using *bot-eq-sup-iff plus-def* **apply** *blast*
apply *simp*
apply *simp*
using *times-inf* **apply** *presburger*
by *simp*

end

class *linorder-stone-kleene-relation-algebra-plus-expansion* =
linorder-stone-kleene-relation-algebra-expansion +
linorder-stone-relation-algebra-plus-expansion
begin

subclass *linear-aggregation-kleene-algebra*
  apply *unfold-locales*
  by *simp*

end

class *linorder-stone-kleene-relation-algebra-tarski-consistent-plus-expansion* =
linorder-stone-kleene-relation-algebra-tarski-consistent-expansion +
linorder-stone-kleene-relation-algebra-plus-expansion

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*)
unbundle *no uminus-syntax*

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* ⊓ *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*)
| *less-eq-ext* (*Val* -) *Top* = *True*
| *less-eq-ext Top Bot* = *False*
| *less-eq-ext Top* (*Val* -) = *False*
| *less-eq-ext Top Top* = *True*

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 add*: *add.assoc*)
  show $x + y = y + x$
    by (*cases x*; *cases y*) (*simp-all add*: *add.commute*)
  show $(x < y) = (x \le y \wedge \neg\, y \le x)$
    by *simp*
  show $x \le x$
    using *less-eq-ext.elims(3)* **by** *fastforce*
  show $x \le y \Longrightarrow y \le z \Longrightarrow x \le z$
    by (*cases x*; *cases y*; *cases z*) *simp-all*
  show $x \le y \Longrightarrow y \le x \Longrightarrow x = y$
    by (*cases x*; *cases y*) *simp-all*
  show $x \sqcap y \le x$
    by (*cases x*; *cases y*) *simp-all*
  show $x \sqcap y \le y$
    by (*cases x*; *cases y*) *simp-all*
  show $x \le y \Longrightarrow x \le z \Longrightarrow x \le y \sqcap z$
    by (*cases x*; *cases y*; *cases z*) *simp-all*
  show $x \le x \sqcup y$
    by (*cases x*; *cases y*) *simp-all*
  show $y \le x \sqcup y$
    by (*cases x*; *cases y*) *simp-all*
  show $y \le x \Longrightarrow z \le x \Longrightarrow y \sqcup z \le x$
    by (*cases x*; *cases y*; *cases z*) *simp-all*
  show $bot \le x$
    by (*simp add*: *bot-ext-def*)
  show $x \le top$
    by (*cases x*) (*simp-all add*: *top-ext-def*)
  show $x \neq bot \wedge x + bot \le y + bot \longrightarrow x + z \le y + z$
    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 $x + y = bot \longrightarrow x = bot$
    by (*cases x*; *cases y*) (*simp-all add*: *bot-ext-def*)
  show $x \le y \vee y \le x$
    by (*cases x*; *cases y*) (*simp-all add*: *linear*)
  show $-x = (\mathit{if}\ x = bot\ \mathit{then}\ top\ \mathit{else}\ bot)$
    by (*cases x*) (*simp-all add*: *bot-ext-def top-ext-def*)
  show $(1{::}'a\ ext) = top$
    by (*simp add*: *one-ext-def top-ext-def*)
  show $x * y = x \sqcap y$
    by *simp*
  show $x^T = x$
    by *simp*
  show $x^\star = top$
    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 :: ($'a$::*enum*,*real ext*) *square*
  shows $minarc_M\ x \preceq \ominus\ominus x$
  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 $x\ y\ z$ :: *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 $(x < y) = (x \leq y \wedge \neg\ y \leq x)$
    by (*cases* x; *cases* y) *auto*
  show $x \leq x$
    by (*cases* x) *simp*
  show $x \leq y \Longrightarrow y \leq z \Longrightarrow x \leq z$
    by (*cases* x; *cases* y; *cases* z) *simp*
  show $x \leq y \Longrightarrow y \leq x \Longrightarrow x = y$
    by (*cases* x; *cases* y) *simp*
  show $x \leq y \Longrightarrow z + x \leq z + y$
    by (*cases* x; *cases* y; *cases* z) *simp*
  show $x \leq y \vee y \leq x$
    by (*cases* x; *cases* y) *auto*
qed

end

lemma *example-real-max-ext-matrix*:
  fixes x :: ($'a$::*enum*,*real-max ext*) *square*
  shows $minarc_M\ x \preceq \ominus\ominus x$
  by (*rule agg-square-m-algebra.minarc-below*)

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*) = ($x \leq y$)
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 $(x < y) = (x \leq y \wedge \neg\, y \leq x)$
    by (*cases x*; *cases y*) *auto*
  show $x \leq x$
    by (*cases x*) *simp*
  show $x \leq y \Longrightarrow y \leq z \Longrightarrow x \leq z$
    by (*cases x*; *cases y*; *cases z*) *simp*
  show $x \leq y \Longrightarrow y \leq x \Longrightarrow x = y$
    by (*cases x*; *cases y*) *simp*
  show $x \leq y \Longrightarrow z + x \leq z + y$
    by (*cases x*; *cases y*; *cases z*) *simp*
  show $x \leq y \vee y \leq x$
    by (*cases x*; *cases y*) *auto*
qed

end

lemma *example-real-min-ext-matrix*:
  fixes *x* :: (*'a::enum,real-min ext*) *square*
  shows $minarc_M\ x \preceq \ominus\ominus x$
  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 $\bot$).

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* ⇒ *'a ext0* ⇒ *'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* ⇒ *'a ext0* ⇒ *'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* ⇒ *'a ext0* ⇒ *'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* ⇒ *'a ext0* ⇒ *'a ext0* **where** *times-ext0 x y* = *x* ⊓ *y*

fun *uminus-ext0* :: *'a ext0* ⇒ *'a ext0* **where**
  uminus-ext0 Bot = *Top*
| *uminus-ext0* (*Val* -) = *Bot*
| *uminus-ext0 Top* = *Bot*

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*) = ($x \leq y$)
| *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* = ($x \leq y \wedge \neg\, y \leq x$)

instance
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 $(x < y) = (x \leq y \wedge \neg\, y \leq x)$
    by *simp*
  show $x \leq x$
    using *less-eq-ext0.elims*(*3*) **by** *fastforce*
  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 $x \sqcap y \leq x$
    by (*cases x*; *cases y*) *simp-all*
  show $x \sqcap y \leq y$
    by (*cases x*; *cases y*) *simp-all*
  show $x \leq y \Longrightarrow x \leq z \Longrightarrow x \leq y \sqcap z$
    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 (*simp add*: *bot-ext0-def*)
  show $x \leq$ *top*
    by (*cases x*) (*simp-all add*: *top-ext0-def*)
  show $x \neq bot \wedge x + bot \leq y + bot \longrightarrow x + z \leq y + z$
    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 $x + y + bot = x + y$
    by (*cases x*; *cases y*) (*simp-all add*: *bot-ext0-def*)

show $x + y = bot \longrightarrow x = bot$
  by (*cases x*; *cases y*) (*simp-all add*: *bot-ext0-def*)
show $x \leq y \vee y \leq x$
  by (*cases x*; *cases y*) (*simp-all add*: *linear*)
show $-x = (\textit{if } x = bot \textit{ then } top \textit{ else } bot)$
  by (*cases x*) (*simp-all add*: *bot-ext0-def top-ext0-def*)
show $(1::'a\ ext0) = top$
  by (*simp add*: *one-ext0-def top-ext0-def*)
show $x * y = x \sqcap y$
  by *simp*
show $x^T = x$
  by *simp*
show $x^\star = top$
  by (*simp add*: *top-ext0-def*)
qed

end

An example of a linearly ordered commutative monoid is the set of real numbers with standard addition and unit 0.

instantiation *real* :: *linordered-comm-monoid-add*
begin

instance ..

end

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*]: $bot + x = x$
begin

sublocale *linordered-comm-monoid-add* **where** $zero = bot$
  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* ⊓ *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* ≤ *y* ∧ ¬ *y* ≤ *x*)
    by *simp*

show $x \le x$
by (*cases* x) *simp-all*
show $x \le y \Longrightarrow y \le z \Longrightarrow x \le z$
by (*cases* x; *cases* y; *cases* z) *simp-all*
show $x \le y \Longrightarrow y \le x \Longrightarrow x = y$
by (*cases* x; *cases* y) *simp-all*
show $x \sqcap y \le x$
by (*cases* x; *cases* y) *simp-all*
show $x \sqcap y \le y$
by (*cases* x; *cases* y) *simp-all*
show $x \le y \Longrightarrow x \le z \Longrightarrow x \le y \sqcap z$
by (*cases* x; *cases* y; *cases* z) *simp-all*
show $x \le x \sqcup y$
by (*cases* x; *cases* y) *simp-all*
show $y \le x \sqcup y$
by (*cases* x; *cases* y) *simp-all*
show $y \le x \Longrightarrow z \le x \Longrightarrow y \sqcup z \le x$
by (*cases* x; *cases* y; *cases* z) *simp-all*
show *bot* $\le x$
by (*cases* x) (*simp-all add*: *bot-extT-def*)
show $x \le$ *top*
by (*cases* x) (*simp-all add*: *top-extT-def*)
show $x \ne bot \wedge x + bot \le y + bot \longrightarrow x + z \le y + z$
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 $x \le y \vee y \le x$
by (*cases* x; *cases* y) (*simp-all add*: *linear*)
show $-x = (\textit{if } x = bot \textit{ then } top \textit{ else } bot)$
by (*cases* x) (*simp-all add*: *bot-extT-def top-extT-def*)
show (*1*::'*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 $x^\star = top$
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 ≤ 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 ≤ 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 bot ≤ x
    by (cases x) (simp-all add: bot-real-max-bot-def)
  show 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* ≤ *x*
  using *add-left-mono top.extremum* **by** *fastforce*

lemma *t-min*: *x* + *y* ≤ *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* = $x \sqcap y$

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*) = $(x \le y)$

fun *less-extB* :: *'a extB* ⇒ *'a extB* ⇒ *bool* **where** *less-extB x y* = $(x \le y \wedge \neg\, y \le x)$

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 $(x < y) = (x \le y \wedge \neg\, y \le x)$
by *simp*
show $x \le x$
by (*cases x*) *simp-all*
show $x \le y \Longrightarrow y \le z \Longrightarrow x \le z$
by (*cases x*; *cases y*; *cases z*) *simp-all*
show $x \le y \Longrightarrow y \le x \Longrightarrow x = y$
by (*cases x*; *cases y*) *simp-all*
show $x \sqcap y \le x$
by (*cases x*; *cases y*) *simp-all*
show $x \sqcap y \le y$
by (*cases x*; *cases y*) *simp-all*
show $x \le y \Longrightarrow x \le z \Longrightarrow x \le y \sqcap z$
by (*cases x*; *cases y*; *cases z*) *simp-all*
show $x \le x \sqcup y$
by (*cases x*; *cases y*) *simp-all*
show $y \le x \sqcup y$
by (*cases x*; *cases y*) *simp-all*
show $y \le x \Longrightarrow z \le x \Longrightarrow y \sqcup z \le x$
by (*cases x*; *cases y*; *cases z*) *simp-all*

show *bot* $\leq x$
by (*simp add*: *bot-extB-def*)
show *1*: $x \leq$ *top*
by (*cases x*) (*simp-all add*: *top-extB-def*)
show $x \neq$ *bot* $\wedge$ $x +$ *bot* $\leq y +$ *bot* $\longrightarrow x + z \leq y + z$
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*(*3*) *top-zero*)
by (*simp-all add*: *bot-extB-def add-right-mono*)
show $x + y +$ *bot* $= x + y$
by (*cases x*; *cases y*) (*simp-all add*: *bot-extB-def*)
show $x + y =$ *bot* $\longrightarrow x =$ *bot*
by (*cases x*; *cases y*) (*simp-all add*: *bot-extB-def*)
show $x \leq y \vee y \leq x$
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-extB-def top-extB-def*)
show (*1*::*'a extB*) $=$ *top*
by (*simp add*: *one-extB-def top-extB-def*)
show $x * y = x \sqcap y$
by *simp*
show $x^T = x$
by *simp*
show $x^\star =$ *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*) = $(x \leq y)$

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 instance 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*
  using *bot-unit.rep-eq top-unit.rep-eq less-eq-unit.rep-eq less-unit.rep-eq unit.Rep-unit-inject unit.Rep-unit* **by** *auto*

end

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

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

abbreviation *tl* :: *real* ⇒ *real* ⇒ *real* **where**
  $tl\ x\ y \equiv max\ (x + y - 1)\ 0$

lemma *tl-assoc*:
  $x \in \{0..1\} \Longrightarrow z \in \{0..1\} \Longrightarrow tl\ (tl\ x\ y)\ z = tl\ x\ (tl\ y\ z)$
  by *auto*

lemma *tl-top-zero*:
  $x \in \{0..1\} \Longrightarrow tl\ 1\ x = x$
  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*
  apply (*simp add*: *less-eq-unit.rep-eq plus-unit.rep-eq*)
  by (*metis* (*mono-tags*, *lifting*) *top-unit.rep-eq unit.Rep-unit-inject unit.Rep-unit plus-unit.rep-eq tl-top-zero*)

end

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** *conv* = *id* **and** *one* = *top* **and** *star* = λx . *top* **and** *uminus* = λx . *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* ≤ *x* + *y*
  using *add-left-mono bot.extremum* **by** *fastforce*

lemma *t-max*: *max x y* ≤ *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**
  $sp\ x\ y \equiv x + y - x * y$

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 $z \in \{0..1\}$
    and $x \le y$
   shows $sp\ z\ x \le sp\ z\ y$
proof −
  have $z + (1 - z) * x \le z + (1 - z) * y$
   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*
  assume *1*: $x \in \{0..1\}$
  assume *2*: $y \in \{0..1\}$
  have $x - x * y \le 1 - y$
   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*: $x + y - x * y \le 1$
   by *simp*
  have $y * (x - 1) \le 0$
   using *1 2* **by** (*meson atLeastAtMost-iff le-iff-diff-le-0 mult-nonneg-nonpos*)

hence $x + y - x * y \geq 0$
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 $x + y - x * y \in \{0..1\}$
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* $\Rightarrow$ $'a$ *extC* $\Rightarrow$ $'a$ *extC* **where** *times-extC* x y $= x \sqcap y$

fun *uminus-extC* :: $'a$ *extC* $\Rightarrow$ $'a$ *extC* **where**
  uminus-extC Bot = *Top*
| *uminus-extC* (*Val* -) = *Bot*
| *uminus-extC Top* = *Bot*

fun *star-extC* :: $'a$ *extC* $\Rightarrow$ $'a$ *extC* **where** *star-extC* - = *Top*

fun *conv-extC* :: $'a$ *extC* $\Rightarrow$ $'a$ *extC* **where** *conv-extC* $x = x$

definition *bot-extC* :: $'a$ *extC* **where** *bot-extC* $\equiv$ *Bot*
definition *one-extC* :: $'a$ *extC* **where** *one-extC* $\equiv$ *Top*
definition *top-extC* :: $'a$ *extC* **where** *top-extC* $\equiv$ *Top*

fun *less-eq-extC* :: $'a$ *extC* $\Rightarrow$ $'a$ *extC* $\Rightarrow$ *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* $\Rightarrow$ $'a$ *extC* $\Rightarrow$ *bool* **where** *less-extC* x y $= (x \leq y \wedge \neg\, y \leq x)$

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 $(x < y) = (x \leq y \wedge \neg\, y \leq x)$
    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 $x \sqcap y \leq x$
    by (*cases* x; *cases* y) *simp-all*
  show $x \sqcap y \leq y$
    by (*cases* x; *cases* y) *simp-all*
  show $x \leq y \Longrightarrow x \leq z \Longrightarrow x \leq y \sqcap z$
    by (*cases* x; *cases* y; *cases* z) *simp-all*

show $x \le x \sqcup y$
by (*cases* x; *cases* y) *simp-all*
show $y \le x \sqcup y$
by (*cases* x; *cases* y) *simp-all*
show $y \le x \Longrightarrow z \le x \Longrightarrow y \sqcup z \le x$
by (*cases* x; *cases* y; *cases* z) *simp-all*
show $bot \le x$
by (*simp add*: *bot-extC-def*)
show $x \le top$
by (*cases* x) (*simp-all add*: *top-extC-def*)
show $x \neq bot \wedge x + bot \le y + bot \longrightarrow x + z \le y + z$
by *simp*
show $x + y + bot = x + y$
by *simp*
show $x + y = bot \longrightarrow x = bot$
by (*simp add*: *bot-extC-def*)
show $x \le y \vee y \le x$
by (*cases* x; *cases* y) (*simp-all add*: *linear*)
show $-x = (\textit{if } x = bot \textit{ then } top \textit{ else } bot)$
by (*cases* x) (*simp-all add*: *bot-extC-def top-extC-def*)
show $(1::'a\ extC) = top$
by (*simp add*: *one-extC-def top-extC-def*)
show $x * y = x \sqcap y$
by *simp*
show $x^T = x$
by *simp*
show $x^\star = top$
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

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* :: $'a::ord \Rightarrow 'a \Rightarrow 'a \Rightarrow 'a$ **where**
median3 $x\ y\ z =$
$(\textit{if } x \le y \wedge y \le z \textit{ then } y \textit{ else}$
$\textit{if } x \le z \wedge z \le y \textit{ then } z \textit{ else}$

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*
| *plus-extN* (*Val -*) (*Val -*) = *N 2*
| *plus-extN* (*Val -*) (*N y*) = *N* (*y* + *1*)
| *plus-extN* (*Val -*) *Top* = *N 2*
| *plus-extN* (*N x*) *Bot* = *N x*
| *plus-extN* (*N x*) (*Val -*) = *N* (*x* + *1*)
| *plus-extN* (*N x*) (*N y*) = *N* (*x* + *y*)
| *plus-extN* (*N x*) *Top* = *N* (*x* + *1*)
| *plus-extN Top Bot* = *N 1*
| *plus-extN Top* (*Val -*) = *N 2*

| *plus-extN Top* (*N y*) = *N* (*y* + *1*)
| *plus-extN Top Top* = *N 2*

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

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

fun *times-extN* :: *'a extN* ⇒ *'a extN* ⇒ *'a extN* **where** *times-extN x y* = *x* ⊓ *y*

fun *uminus-extN* :: *'a extN* ⇒ *'a extN* **where**
uminus-extN Bot = *Top*
| *uminus-extN* (*Val* -) = *Bot*
| *uminus-extN* (*N* -) = *Bot*
| *uminus-extN Top* = *Bot*

fun *star-extN* :: *'a extN* ⇒ *'a extN* **where** *star-extN* - = *Top*

fun *conv-extN* :: *'a extN* ⇒ *'a extN* **where** *conv-extN x* = *x*

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

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

| *less-eq-extN* (*N* -) *Bot* = *False*
| *less-eq-extN* (*N* -) (*Val* -) = *False*
| *less-eq-extN* (*N* x) (*N* y) = ($x \leq y$)
| *less-eq-extN* (*N* -) *Top* = *True*
| *less-eq-extN* *Top* *Bot* = *False*
| *less-eq-extN* *Top* (*Val* -) = *False*
| *less-eq-extN* *Top* (*N* -) = *False*
| *less-eq-extN* *Top* *Top* = *True*

fun *less-extN* :: 'a *extN* ⇒ 'a *extN* ⇒ *bool* **where** *less-extN* x y = ($x \leq y \wedge \neg\, y \leq x$)

instance
proof
  fix x y z :: 'a *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 $(x < y) = (x \leq y \wedge \neg\, y \leq x)$
    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 $x \sqcap y \leq x$
    by (*cases* x; *cases* y) *simp-all*
  show $x \sqcap y \leq y$
    by (*cases* x; *cases* y) *simp-all*
  show $x \leq y \Longrightarrow x \leq z \Longrightarrow x \leq y \sqcap z$
    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 (*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 $x \le y \lor y \le x$
  by (*cases* x; *cases* y) (*simp-all add*: *linear*)
show $-x = (\textit{if } x = \textit{bot then top else bot})$
  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 $x * y = x \sqcap y$
  by *simp*
show $x^T = x$
  by *simp*
show $x^\star = top$
  by (*simp add*: *top-extN-def*)
qed

end

end

6 An Operation to Select Components in Algebras with Minimisation

In this theory we show that an operation to select components of a graph can be defined in m-Kleene Algebras. This work is by Nicolas Robinson-O'Brien.

theory *M-Choose-Component*

imports
  Stone-Relation-Algebras.Choose-Component
  Matrix-Aggregation-Algebras

begin

Every *m-kleene-algebra* is an instance of *choose-component-algebra* when the *choose-component* operation is defined as follows:

context *m-kleene-algebra*
begin

definition *m-choose-component* $e\ v \equiv$
  if vector-classes $e\ v$ *then*
    $e * minarc(v) * top$
  else
    bot

sublocale *m-choose-component-algebra*: *choose-component-algebra* **where**
choose-component = *m-choose-component*
proof
  fix $e\ v$

show *m-choose-component e v* $\leq --\ v$
proof (*cases vector-classes e v*)
case *True*
hence *m-choose-component e v* $= e * minarc(v) * top$
by (*simp add*: *m-choose-component-def*)
also have ... $\leq e * --v * top$
by (*simp add*: *comp-isotone minarc-below*)
also have ... $= e * v * top$
using *True vector-classes-def* **by** *auto*
also have ... $\leq v * top$
using *True vector-classes-def mult-assoc* **by** *auto*
finally show *?thesis*
using *True vector-classes-def* **by** *auto*
next
case *False*
hence *m-choose-component e v* $= bot$
using *False m-choose-component-def* **by** *auto*
thus *?thesis*
by *simp*
qed
next
fix *e v*
show *vector* (*m-choose-component e v*)
proof (*cases vector-classes e v*)
case *True*
thus *?thesis*
by (*simp add*: *mult-assoc m-choose-component-def*)
next
case *False*
thus *?thesis*
by (*simp add*: *m-choose-component-def*)
qed
next
fix *e v*
show *regular* (*m-choose-component e v*)
using *minarc-regular regular-mult-closed vector-classes-def m-choose-component-def* **by** *auto*
next
fix *e v*
show *m-choose-component e v* $*$ (*m-choose-component e v*)$^T \leq e$
proof (*cases vector-classes e v*)
case *True*
assume *1*: *vector-classes e v*
hence *m-choose-component e v* $*$ (*m-choose-component e v*)$^T = e * minarc(v) * top * (e * minarc(v) * top)^T$
by (*simp add*: *m-choose-component-def*)
also have ... $= e * minarc(v) * top * top^T * minarc(v)^T * e^T$
by (*metis comp-associative conv-dist-comp*)
also have ... $= e * minarc(v) * top * top * minarc(v)^T * e$

using *True vector-classes-def* **by** *auto*
also have ... = $e * minarc(v) * top * minarc(v)^T * e$
by (*simp add*: *comp-associative*)
also have ... $\leq e$
proof (*cases* $v = bot$)
case *True*
thus *?thesis*
by (*simp add*: *True minarc-bot*)
next
case *False*
assume *3*: $v \neq bot$
hence $e * minarc(v) * top * minarc(v)^T \leq e * 1$
using *3 minarc-arc arc-expanded comp-associative mult-right-isotone* **by** *fastforce*
hence $e * minarc(v) * top * minarc(v)^T * e \leq e * 1 * e$
using *mult-left-isotone* **by** *auto*
also have ... = e
using *True preorder-idempotent vector-classes-def* **by** *auto*
thus *?thesis*
using *calculation* **by** *auto*
qed
thus *?thesis*
by (*simp add*: *calculation*)
next
case *False*
thus *?thesis*
by (*simp add*: *m-choose-component-def*)
qed
next
fix *e v*
show $e *$ *m-choose-component e v* $\leq$ *m-choose-component e v*
proof (*cases vector-classes e v*)
case *True*
thus *?thesis*
using *comp-right-one dual-order.eq-iff mult-isotone vector-classes-def m-choose-component-def mult-assoc* **by** *metis*
next
case *False*
thus *?thesis*
by (*simp add*: *m-choose-component-def*)
qed
next
fix *e v*
show *vector-classes e v* $\longrightarrow$ *m-choose-component e v* $\neq bot$
proof (*cases vector-classes e v*)
case *True*
hence *m-choose-component e v* $\geq minarc(v) * top$
using *vector-classes-def m-choose-component-def comp-associative minarc-arc shunt-bijective* **by** *fastforce*

also have ... $\geq$ *minarc*(v)
using *calculation dual-order.trans top-right-mult-increasing* **by** *blast*
thus *?thesis*
using *le-bot minarc-bot-iff vector-classes-def* **by** *fastforce*
next
case *False*
thus *?thesis*
by *blast*
qed
qed

sublocale *m-choose-component-algebra-tarski*: *choose-component-algebra-tarski*
where *choose-component* = *m-choose-component*
..

end

class *m-kleene-algebra-choose-component* = *m-kleene-algebra* +
choose-component-algebra

end

References

[1] W. Guttmann. Relation-algebraic verification of Prim's minimum spanning tree algorithm. In A. Sampaio and F. Wang, editors, *Theoretical Aspects of Computing – ICTAC 2016*, volume 9965 of *Lecture Notes in Computer Science*, pages 51–68. Springer, 2016.

[2] W. Guttmann. Stone-Kleene relation algebras. *Archive of Formal Proofs*, 2017.

[3] W. Guttmann. Stone relation algebras. In P. Höfner, D. Pous, and G. Struth, editors, *Relational and Algebraic Methods in Computer Science*, volume 10226 of *Lecture Notes in Computer Science*, pages 127–143. Springer, 2017.

[4] W. Guttmann. An algebraic framework for minimum spanning tree problems. *Theoretical Computer Science*, 2018. https://doi.org/10.1016/j.tcs.2018.04.012.

[5] W. Guttmann. Verifying minimum spanning tree algorithms with Stone relation algebras. *Journal of Logical and Algebraic Methods in Programming*, 2018. https://doi.org/10.1016/j.jlamp.2018.09.005.

[6] W. Guttmann and N. Robinson-O'Brien. Relational minimum spanning tree algorithms. *Archive of Formal Proofs*, 2020.