# Aggregation Algebras

Walter Guttmann

September 23, 2021

**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].

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*
  ⟨*proof*⟩

**interpretation** *comp?*: *comp-fun-commute* $f \circ g$
  ⟨*proof*⟩

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

**lemma** *infinite* [*simp*]: $\neg\ finite\ A \Longrightarrow F\ g\ A = \mathbf{1}$
  ⟨*proof*⟩

**lemma** *empty* [*simp*]: $F\ g\ \{\} = \mathbf{1}$
  ⟨*proof*⟩

**lemma** *insert* [*simp*]: $finite\ A \Longrightarrow x \notin A \Longrightarrow F\ g\ (insert\ x\ A) = g\ x * F\ g\ A$
  ⟨*proof*⟩

**lemma** *remove*:
  **assumes** *finite A* **and** $x \in A$
  **shows** $F\ g\ A = g\ x * F\ g\ (A - \{x\})$
⟨*proof*⟩

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

**lemma** *insert-if*: $finite\ A \Longrightarrow F\ g\ (insert\ x\ A) = (if\ x \in A\ then\ F\ g\ A\ else\ g\ x * F\ g\ A)$
  ⟨*proof*⟩

**lemma** *neutral*: $\forall x \in A.\ g\ x = \mathbf{1} \Longrightarrow F\ g\ A = \mathbf{1}$
  ⟨*proof*⟩

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

⟨*proof*⟩

**lemma** *F-one* [*simp*]: $F\ g\ A * \mathbf{1} = F\ g\ A$
⟨*proof*⟩

**lemma** *one-F* [*simp*]: $\mathbf{1} * F\ g\ A = F\ g\ A$
⟨*proof*⟩

**lemma** *F-g-one* [*simp*]: $F\ (\lambda x\ .\ g\ x * \mathbf{1})\ A = F\ g\ A$
⟨*proof*⟩

**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!
⟨*proof*⟩

**corollary** *union-inter-neutral*:
**assumes** *finite* $A$ **and** *finite* $B$
**and** $\forall x \in A \cap B.\ g\ x = \mathbf{1}$
**shows** $F\ g\ (A \cup B) = F\ g\ A * F\ g\ B$
⟨*proof*⟩

**corollary** *union-disjoint*:
**assumes** *finite* $A$ **and** *finite* $B$
**assumes** $A \cap B = \{\}$
**shows** $F\ g\ (A \cup B) = F\ g\ A * F\ g\ B$
⟨*proof*⟩

**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*⟩

**lemma** *subset-diff*:
**assumes** $B \subseteq A$ **and** *finite* $A$
**shows** $F\ g\ A = F\ g\ (A - B) * F\ g\ B$
⟨*proof*⟩

**lemma** *setdiff-irrelevant*:
**assumes** *finite* $A$
**shows** $F\ g\ (A - \{x.\ g\ x = z\}) = F\ g\ A$
⟨*proof*⟩

**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*⟩

**lemma** *reindex*:
**assumes** *inj-on h A*
**shows** *F g (h ‘ A) = F (g ∘ h) A*
⟨*proof*⟩

**lemma** *cong* [*fundef-cong*]:
**assumes** *A = B*
**assumes** *g-h*: *⋀x. x ∈ B ⟹ g x = h x*
**shows** *F g A = F h B*
⟨*proof*⟩

**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*
⟨*proof*⟩

**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*
⟨*proof*⟩

**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*
⟨*proof*⟩

**lemma** *Union-disjoint*:
**assumes** *∀ A∈C. finite A ∀ A∈C. ∀ B∈C. A ≠ B ⟶ A ∩ B = {}*
**shows** *F g (⋃ C) = (F ∘ F) g C*
⟨*proof*⟩

**lemma** *distrib*: *F (λx. g x ∗ h x) A = F g A ∗ F h A*
⟨*proof*⟩

**lemma** *Sigma*:
*finite A ⟹ ∀ x∈A. finite (B x) ⟹ F (λx. F (g x) (B x)) A = F (case-prod g) (SIGMA x:A. B x)*
⟨*proof*⟩

**lemma** *related*:
**assumes** *Re*: *R* **1** **1**
**and** *Rop*: *∀ x1 y1 x2 y2. R x1 x2 ∧ R y1 y2 ⟶ R (x1 ∗ y1) (x2 ∗ y2)*
**and** *fin*: *finite S*
**and** *R-h-g*: *∀ x∈S. R (h x) (g x)*
**shows** *R (F h S) (F g S)*
⟨*proof*⟩

**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*⟩

**lemma** *mono-neutral-cong-right*:
*finite* $T \Longrightarrow S \subseteq T \Longrightarrow \forall i \in T - S.\ g\ i = \mathbf{1} \Longrightarrow (\bigwedge x.\ x \in S \Longrightarrow g\ x = h\ x)$
$\Longrightarrow$
$F\ g\ T = F\ h\ S$
⟨*proof*⟩

**lemma** *mono-neutral-left*: *finite* $T \Longrightarrow S \subseteq T \Longrightarrow \forall i \in T - S.\ g\ i = \mathbf{1} \Longrightarrow F\ g\ S = F\ g\ T$
⟨*proof*⟩

**lemma** *mono-neutral-right*: *finite* $T \Longrightarrow S \subseteq T \Longrightarrow \forall i \in T - S.\ g\ i = \mathbf{1} \Longrightarrow F\ g\ T = F\ g\ S$
⟨*proof*⟩

**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*⟩

**lemma** *reindex-bij-betw*: *bij-betw h S T* $\Longrightarrow F\ (\lambda x.\ g\ (h\ x))\ S = F\ g\ T$
⟨*proof*⟩

**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*⟩

**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*⟩

**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*⟩

**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*⟩

**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*⟩

**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* $\mathbf{1}$)
⟨*proof*⟩

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

**lemma** *If-cases*:
**fixes** $P :: 'b \Rightarrow bool$ **and** $g\ h :: 'b \Rightarrow 'a$
**assumes** *fin*: *finite A*
**shows** $F\ (\lambda x.$ *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*⟩

**lemma** *cartesian-product*: $F\ (\lambda x.\ F\ (g\ x)\ B)\ A = F\ (case\text{-}prod\ g)\ (A \times B)$

⟨*proof*⟩

**lemma** *inter-restrict*:
**assumes** *finite A*
**shows** $F\ g\ (A \cap B) = F\ (\lambda x.\ \textit{if}\ x \in B\ \textit{then}\ g\ x\ \textit{else}\ \mathbf{1})\ A$
⟨*proof*⟩

**lemma** *inter-filter*:
$\textit{finite}\ A \Longrightarrow F\ g\ \{x \in A.\ P\ x\} = F\ (\lambda x.\ \textit{if}\ P\ x\ \textit{then}\ g\ x\ \textit{else}\ \mathbf{1})\ A$
⟨*proof*⟩

**lemma** *Union-comp*:
**assumes** $\forall A \in B.\ \textit{finite}\ A$
**and** $\bigwedge A1\ A2\ x.\ A1 \in B \Longrightarrow A2 \in B \Longrightarrow A1 \neq A2 \Longrightarrow x \in A1 \Longrightarrow x \in A2 \Longrightarrow g\ x = \mathbf{1}$
**shows** $F\ g\ (\bigcup B) = (F \circ F)\ g\ B$
⟨*proof*⟩

**lemma** *swap*: $F\ (\lambda i.\ F\ (g\ i)\ B)\ A = F\ (\lambda j.\ F\ (\lambda i.\ g\ i\ j)\ A)\ B$
⟨*proof*⟩

**lemma** *swap-restrict*:
$\textit{finite}\ A \Longrightarrow \textit{finite}\ B \Longrightarrow$
$F\ (\lambda x.\ F\ (g\ x)\ \{y.\ y \in B \wedge R\ x\ y\})\ A = F\ (\lambda y.\ F\ (\lambda x.\ g\ x\ y)\ \{x.\ x \in A \wedge R\ x\ y\})\ B$
⟨*proof*⟩

**lemma** *Plus*:
**fixes** $A :: 'b\ \textit{set}$ **and** $B :: 'c\ \textit{set}$
**assumes** *fin*: *finite A finite B*
**shows** $F\ g\ (A <+> B) = F\ (g \circ \textit{Inl})\ A * F\ (g \circ \textit{Inr})\ B$
⟨*proof*⟩

**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*⟩

**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** $F\ g\ A = F\ h\ B$
⟨*proof*⟩

**end**

## 2.2 Generalized summation over a set

**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*
  ⟨*proof*⟩

**abbreviation** *Sum-0* ($\sum$)
  **where** $\sum \equiv$ *sum-0* $(\lambda x.\ x)$

**end**

**context** *comm-monoid-add*
**begin**

**subclass** *ab-semigroup-add-0*
  ⟨*proof*⟩

**end**

Now: lots of fancy syntax. First, *sum-0* $(\lambda x.\ e)$ $A$ is written $\sum x{\in}A.\ e$.

**syntax** (*ASCII*)
  *-sum* :: *pttrn* ⇒ *'a set* ⇒ *'b* ⇒ *'b*::*comm-monoid-add* ((*3SUM* (-/:-)./ -) [*0*, *51*, *10*] *10*)
**syntax**
  *-sum* :: *pttrn* ⇒ *'a set* ⇒ *'b* ⇒ *'b*::*comm-monoid-add* ((*2*$\sum$ (-/∈-)./ -) [*0*, *51*, *10*] *10*)
**translations** — Beware of argument permutation!
  $\sum i{\in}A.\ b \rightleftharpoons$ *CONST sum-0* $(\lambda i.\ b)$ $A$

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

**syntax** (*ASCII*)
  *-qsum* :: *pttrn* ⇒ *bool* ⇒ *'a* ⇒ *'a* ((*3SUM* - |/ -./ -) [*0*, *0*, *10*] *10*)
**syntax**
  *-qsum* :: *pttrn* ⇒ *bool* ⇒ *'a* ⇒ *'a* ((*2*$\sum$ - | (-)./ -) [*0*, *0*, *10*] *10*)
**translations**
  $\sum x|P.\ t$ => *CONST sum-0* $(\lambda x.\ t)$ $\{x.\ P\}$

⟨*ML*⟩

**lemma** (**in** *ab-semigroup-add-0*) *sum-image-gen-0*:
  **assumes** *fin*: *finite S*
  **shows** *sum-0 g S* = *sum-0* $(\lambda y.$ *sum-0 g* $\{x.\ x \in S \wedge f\ x = y\})$ $(f$ ‘ $S)$
⟨*proof*⟩

### 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*⟩

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

**lemma** *add-nonneg-nonneg* [*simp*]: $0 \leq a \Longrightarrow 0 \leq b \Longrightarrow 0 \leq a + b$
⟨*proof*⟩

**lemma** *add-nonpos-nonpos*: $a \leq 0 \Longrightarrow b \leq 0 \Longrightarrow a + b \leq 0$
⟨*proof*⟩

**end**

**lemma** (**in** *ordered-ab-semigroup-add-0*) *sum-mono*:
$(\bigwedge i.\ i \in K \Longrightarrow f\ i \leq g\ i) \Longrightarrow (\sum i \in K.\ f\ i) \leq (\sum i \in K.\ g\ i)$
⟨*proof*⟩

**lemma** (**in** *ordered-ab-semigroup-add-0*) *sum-mono-00*:
$(\bigwedge i.\ i \in K \Longrightarrow f\ i + 0 \leq g\ i + 0) \Longrightarrow (\sum i \in K.\ f\ i) \leq (\sum i \in K.\ g\ i)$
⟨*proof*⟩

**lemma** (**in** *ordered-ab-semigroup-add-0*) *sum-mono-0*:
$(\bigwedge i.\ i \in K \Longrightarrow f\ i + 0 \leq g\ i) \Longrightarrow (\sum i \in K.\ f\ i) \leq (\sum i \in K.\ g\ i)$
⟨*proof*⟩

**context** *ordered-ab-semigroup-add-0*
**begin**

**lemma** *sum-nonneg*: $(\bigwedge x.\ x \in A \Longrightarrow 0 \leq f\ x) \Longrightarrow 0 \leq$ *sum-0 f A*
⟨*proof*⟩

**lemma** *sum-nonpos*: $(\bigwedge x.\ x \in A \Longrightarrow f\ x \leq 0) \Longrightarrow$ *sum-0 f A* $\leq 0$
⟨*proof*⟩

**lemma** *sum-mono2*:
**assumes** *fin*: *finite B*
**and** *sub*: $A \subseteq B$
**and** *nn*: $\bigwedge b.\ b \in B - A \Longrightarrow 0 \leq f\ b$
**shows** *sum-0 f A* $\leq$ *sum-0 f B*
⟨*proof*⟩

**lemma** *sum-le-included*:
**assumes** *finite s finite t*

**and** $\forall\, y{\in}t.\ 0 \leq g\ y\ (\forall\, x{\in}s.\ \exists\, y{\in}t.\ i\ y = x \wedge f\ x \leq g\ y)$
**shows** *sum-0 f s* $\leq$ *sum-0 g t*
⟨*proof*⟩

**end**

**lemma** *sum-comp-morphism*:
$h\ 0 = 0 \Longrightarrow (\bigwedge x\ y.\ h\ (x + y) = h\ x + h\ y) \Longrightarrow$ *sum-0* $(h \circ g)\ A = h$ (*sum-0 g A*)
⟨*proof*⟩

**lemma** *sum-cong-Suc*:
**assumes** $0 \notin A\ \bigwedge x.\ Suc\ x \in A \Longrightarrow f\ (Suc\ x) = g\ (Suc\ x)$
**shows** *sum-0 f A* = *sum-0 g A*
⟨*proof*⟩

**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 $\leq$-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** $sum\ ((x \sqcup y) \sqcap z) = sum\ (x \sqcap z) + sum\ (y \sqcap z)$
⟨*proof*⟩

**lemma** *sum-disjoint-3*:
**assumes** $w \sqcap x = bot$
**and** $w \sqcap y = bot$
**and** $x \sqcap y = bot$
**shows** $sum\ ((w \sqcup x \sqcup y) \sqcap z) = sum\ (w \sqcap z) + sum\ (x \sqcap z) + sum\ (y \sqcap z)$
⟨*proof*⟩

**lemma** *sum-symmetric*:
**assumes** $y = y^T$
**shows** $sum\ (x^T \sqcap y) = sum\ (x \sqcap y)$
⟨*proof*⟩

**lemma** *sum-commute*:

*sum x + sum y = sum y + sum x*
⟨*proof*⟩

**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* ⇒ *'a*

**class** *m-algebra* = *s-algebra* + *minarc* +
**assumes** *minarc-below*: *minarc x* ≤ *−−x*
**assumes** *minarc-arc*: *x* ≠ *bot* ⟶ *arc* (*minarc x*)
**assumes** *minarc-min*: *arc y* ∧ *y* ⊓ *x* ≠ *bot* ⟶ *sum* (*minarc x* ⊓ *x*) ≤ *sum* (*y* ⊓ *x*)
**assumes** *sum-linear*: *sum x* ≤ *sum y* ∨ *sum y* ≤ *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*⟩

**lemma** *minarc-bot*:
*minarc bot* = *bot*
⟨*proof*⟩

**lemma** *minarc-bot-iff*:
*minarc x* = *bot* ⟷ *x* = *bot*
⟨*proof*⟩

**lemma** *minarc-meet-bot*:
**assumes** *minarc x* ⊓ *x* = *bot*
**shows** *minarc x* = *bot*
⟨*proof*⟩

**lemma** *minarc-meet-bot-minarc-iff*:

$minarc\ x \sqcap x = bot \longleftrightarrow minarc\ x = bot$
⟨*proof*⟩

**lemma** *minarc-meet-bot-iff*:
$minarc\ x \sqcap x = bot \longleftrightarrow x = bot$
⟨*proof*⟩

**lemma** *minarc-regular*:
$regular\ (minarc\ x)$
⟨*proof*⟩

**lemma** *minarc-selection*:
$selection\ (minarc\ x \sqcap y)\ y$
⟨*proof*⟩

**lemma** *minarc-below-regular*:
$regular\ x \Longrightarrow minarc\ x \leq x$
⟨*proof*⟩

**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 commu-

tative 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*)
**and** *uminus* (− - [*81*] *80*)

## 4.1 Aggregation Orders and Finite Sums

An aggregation order is a partial order with a least element and an associative commutative operation satisfying certain properties. Axiom *add-add-bot* introduces almost a commutative monoid; we require that *bot* is a unit only on the image of the aggregation operation. This is necessary since it is not a unit of a number of aggregation operations we are interested in. Axiom *add-right-isotone* states that aggregation is ≤-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*
⟨*proof*⟩

**lemma** *add-bot-bot-bot*:
$x + bot + bot + bot = x + bot$
⟨*proof*⟩

**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** (*xsymbols*)

*-sum-ab-semigroup-add-0* :: *idt* ⇒ *'a*::*bounded-semilattice-sup-bot* ⇒ *'a* ((∑- -) [*0*,*10*] *10*)

**translations**
∑*x* *t* => *XCONST ab-semigroup-add-0.sum-0 XCONST plus* (*XCONST plus XCONST bot XCONST bot*) (λ*x* . *t*) { *x* . *CONST True* }

The following are basic properties of such sums.

**lemma** *agg-sum-bot*:
(∑*k* *bot*::*'a*::*aggregation-order*) = *bot* + *bot*
⟨*proof*⟩

**lemma** *agg-sum-bot-bot*:
(∑*k* *bot* + *bot*::*'a*::*aggregation-order*) = *bot* + *bot*
⟨*proof*⟩

**lemma** *agg-sum-not-bot-1*:
**fixes** *f* :: *'a*::*finite* ⇒ *'b*::*aggregation-order*
**assumes** *f i* ≠ *bot*
**shows** (∑*k* *f k*) ≠ *bot*
⟨*proof*⟩

**lemma** *agg-sum-not-bot*:
**fixes** *f* :: (*'a*::*finite*,*'b*::*aggregation-order*) *square*
**assumes** *f* (*i*,*j*) ≠ *bot*
**shows** (∑*k* ∑*l* *f* (*k*,*l*)) ≠ *bot*
⟨*proof*⟩

**lemma** *agg-sum-distrib*:
**fixes** *f g* :: *'a* ⇒ *'b*::*aggregation-order*
**shows** (∑*k* *f k* + *g k*) = (∑*k* *f k*) + (∑*k* *g k*)
⟨*proof*⟩

**lemma** *agg-sum-distrib-2*:
**fixes** *f g* :: (*'a*,*'b*::*aggregation-order*) *square*
**shows** (∑*k* ∑*l* *f* (*k*,*l*) + *g* (*k*,*l*)) = (∑*k* ∑*l* *f* (*k*,*l*)) + (∑*k* ∑*l* *g* (*k*,*l*))
⟨*proof*⟩

**lemma** *agg-sum-add-bot*:
**fixes** *f* :: *'a* ⇒ *'b*::*aggregation-order*
**shows** (∑*k* *f k*) = (∑*k* *f k*) + *bot*
⟨*proof*⟩

**lemma** *agg-sum-add-bot-2*:
**fixes** *f* :: *'a* ⇒ *'b*::*aggregation-order*
**shows** (∑*k* *f k* + *bot*) = (∑*k* *f k*)
⟨*proof*⟩

**lemma** *agg-sum-commute*:

**fixes** $f$ :: ('$a$,'$b$::*aggregation-order*) *square*
**shows** $(\sum_k \sum_l f\ (k,l)) = (\sum_l \sum_k f\ (k,l))$
⟨*proof*⟩

**lemma** *agg-delta*:
**fixes** $f$ :: '$a$::*finite* ⇒ '$b$::*aggregation-order*
**shows** $(\sum_l$ *if* $l = j$ *then* $f\ l$ *else zero*$) = f\ j$ + *bot*
⟨*proof*⟩

**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*⟩

**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*⟩

## 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* = (λ$e$ . *bot* + *bot*)

**definition** *plus-matrix* :: ('$a$,'$b$::*plus*) *square* ⇒ ('$a$,'$b$) *square* ⇒ ('$a$,'$b$) *square* (**infixl** $\oplus_M$ *65*) **where** *plus-matrix* $f\ g$ = (λ$e$ . $f\ e + g\ e$)

**definition** *sum-matrix* :: ('$a$::*enum*,'$b$::{*plus*,*bot*}) *square* ⇒ ('$a$,'$b$) *square* ($sum_M$ - [*80*] *80*) **where** *sum-matrix* $f$ = (λ$(i,j)$ . *if* $i$ = *hd enum-class.enum* ∧ $j = i$ *then* $\sum_k \sum_l f\ (k,l)$ *else bot* + *bot*)

Basic properties of these operations are given in the following.

**lemma** *bot-plus-bot*:
*mbot* $\oplus_M$ *mbot* = *mzero*
⟨*proof*⟩

**lemma** *sum-bot*:
$sum_M$ (*mbot* :: ('$a$::*enum*,'$b$::*aggregation-order*) *square*) = *mzero*
⟨*proof*⟩

**lemma** *sum-plus-bot*:
**fixes** $f :: ('a::enum,'b::aggregation\text{-}order)\ square$
**shows** $sum_M\ f \oplus_M\ mbot = sum_M\ f$
⟨*proof*⟩

**lemma** *sum-plus-zero*:
**fixes** $f :: ('a::enum,'b::aggregation\text{-}order)\ square$
**shows** $sum_M\ f \oplus_M\ mzero = sum_M\ f$
⟨*proof*⟩

**lemma** *agg-matrix-bot*:
**fixes** $f :: ('a,'b::aggregation\text{-}order)\ square$
**assumes** $\forall\, i\ j\ .\ f\ (i,j) = bot$
**shows** $f = mbot$
⟨*proof*⟩

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* $:: ('a,'b::\{plus,bot\})\ square \Rightarrow ('a,'b)\ square$ $(sum2_M$ - [80] 80) **where** *sum-matrix-2* $f = (\lambda e\ .\ \sum_k \sum_l f\ (k,l))$

**lemma** *sum-bot-2*:
$sum2_M\ (mbot :: ('a,'b::aggregation\text{-}order)\ square) = mzero$
⟨*proof*⟩

**lemma** *sum-plus-bot-2*:
**fixes** $f :: ('a,'b::aggregation\text{-}order)\ square$
**shows** $sum2_M\ f \oplus_M\ mbot = sum2_M\ f$
⟨*proof*⟩

**lemma** *sum-plus-zero-2*:
**fixes** $f :: ('a,'b::aggregation\text{-}order)\ square$
**shows** $sum2_M\ f \oplus_M\ mzero = sum2_M\ f$
⟨*proof*⟩

## 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*]: $-x = (\mathit{if}\ x = \mathit{bot}\ \mathit{then}\ \mathit{top}\ \mathit{else}\ \mathit{bot})$
  **assumes** *one-def* [*simp*]: $1 = \mathit{top}$
  **assumes** *times-def* [*simp*]: $x * y = x \sqcap y$
  **assumes** *conv-def* [*simp*]: $x^T = x$
**begin**

**subclass** *stone-algebra*
  ⟨*proof*⟩

**subclass** *stone-relation-algebra*
  ⟨*proof*⟩

**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*⟩

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*⟩

## 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* ⟹ *xs*!(*enum-pos' xs x*) = *x*
  ⟨*proof*⟩

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*
⟨*proof*⟩

**lemma** *enum-pos-injective* [*simp*]:
*enum-pos x* = *enum-pos y* ⟹ *x* = *y*
⟨*proof*⟩

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*
⟨*proof*⟩

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

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

**lemma** *minarc-at-most-one*:
**fixes** *f* :: (*'a::enum*,*'b::*{*aggregation-order*,*top*}) *square*
**assumes** (*minarc*$_M$ *f*) *e* ≠ *bot*
**and** (*minarc*$_M$ *f*) *d* ≠ *bot*
**shows** *e* = *d*
⟨*proof*⟩

## 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*
⟨*proof*⟩

**sublocale** *heyting*: *bounded-heyting-lattice* **where** *implies* = λ*x y* . *if x* ≤ *y then top else y*
⟨*proof*⟩

**end**

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

**lemma** *least-order*:
**assumes** *transitive*: ∀ *x y z* . *le x y* ∧ *le y z* ⟶ *le x z*
**and** *total*: ∀ *x y* . *le x y* ∨ *le y x*
**shows** *finite A* ⟹ *A* ≠ {} ⟹ ∃ *x* . *x* ∈ *A* ∧ (∀ *y* . *y* ∈ *A* ⟶ *le x y*)
⟨*proof*⟩

**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*⟩

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* = (*if x* = *bot then top else bot*)
**assumes** *one-def-2* [*simp*]: *1* = *top*
**assumes** *times-def-2* [*simp*]: *x* ∗ *y* = *x* ⊓ *y*
**assumes** *conv-def-2* [*simp*]: $x^T$ = *x*
**begin**

**subclass** *aggregation-algebra*
⟨*proof*⟩

**lemma** *regular-bot-top-2*:
*regular x* ⟷ *x* = *bot* ∨ *x* = *top*
⟨*proof*⟩

**sublocale** *heyting*: *heyting-stone-algebra* **where** *implies* = λ*x y* . *if x* ≤ *y then top else y*

⟨*proof*⟩

**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*⟩

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*⟩

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

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

**subclass** *stone-kleene-relation-algebra*
  ⟨*proof*⟩

**end**

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

**subclass** *aggregation-kleene-algebra*
  ⟨*proof*⟩

**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* ⟨*proof*⟩

**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* ⟨*proof*⟩

**end**

# 5 Algebras for Aggregation and Minimisation with a Linear Order

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

**theory** *Linear-Aggregation-Algebras*

**imports** *Matrix-Aggregation-Algebras HOL.Real*

**begin**

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

## 5.1 Linearly Ordered Commutative Semigroups

Any linearly ordered commutative semigroup extended by new least and greatest elements forms a linear aggregation lattice. The extension is done so that the new least element is a unit of aggregation and the new greatest element is a zero of aggregation.

**datatype** '*a ext* =
*Bot*
| *Val* '*a*
| *Top*

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

**fun** *plus-ext* :: '*a ext* ⇒ '*a ext* ⇒ '*a ext* **where**
*plus-ext Bot x* = *x*

| *plus-ext* (*Val x*) *Bot* = *Val x*
| *plus-ext* (*Val x*) (*Val y*) = *Val* (*x* + *y*)
| *plus-ext* (*Val* -) *Top* = *Top*
| *plus-ext Top* - = *Top*

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

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

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

**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** $minarc_M\ x \preceq \ominus\ominus x$
⟨*proof*⟩

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*⟩

**end**

**lemma** *example-real-max-ext-matrix*:
**fixes** $x$ :: ('a::*enum*,*real-max ext*) *square*
**shows** $minarc_M\ x \preceq \ominus\ominus x$
⟨*proof*⟩

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*⟩

**end**

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

## 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* ≤ *y*)
| *less-eq-ext0* (*Val -*) *Top* = *True*
| *less-eq-ext0 Top Bot* = *False*
| *less-eq-ext0 Top* (*Val -*) = *False*
| *less-eq-ext0 Top Top* = *True*

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

**instance**
⟨*proof*⟩

**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** ⟨*proof*⟩

**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*

⟨*proof*⟩

**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*⟩

**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*⟩

**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*

⟨*proof*⟩

**lemma** *add-decreasing*: $x + y \leq x$
⟨*proof*⟩

**lemma** *t-min*: $x + y \leq min\ x\ y$
⟨*proof*⟩

**end**

**datatype** *'a extB* =
*Bot*
| *Val 'a*

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

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

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

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

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

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

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

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

**definition** *bot-extB* :: *'a extB* **where** *bot-extB* ≡ *Bot*
**definition** *one-extB* :: *'a extB* **where** *one-extB* ≡ *Val top*
**definition** *top-extB* :: *'a extB* **where** *top-extB* ≡ *Val top*

**fun** *less-eq-extB* :: *'a extB* ⇒ *'a extB* ⇒ *bool* **where**

*less-eq-extB Bot - = True*
| *less-eq-extB (Val -) Bot = False*
| *less-eq-extB (Val x) (Val y) = (x ≤ y)*

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

**instance**
⟨*proof*⟩

**end**

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

**datatype** *real-min-top* =
*R real*
| *PInfty*

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

**definition** *top-real-min-top* ≡ *PInfty*

**fun** *less-eq-real-min-top* **where**
*less-eq-real-min-top - PInfty = True*
| *less-eq-real-min-top PInfty (R -) = False*
| *less-eq-real-min-top (R x) (R y) = (x ≤ y)*

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

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

**instance**
⟨*proof*⟩

**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*
⟨*proof*⟩

**setup-lifting** *type-definition-unit*

**instantiation** *unit* :: *bounded-linorder*
**begin**

**lift-definition** *bot-unit* :: *unit* **is** *0*
⟨*proof*⟩

**lift-definition** *top-unit* :: *unit* **is** *1*
⟨*proof*⟩

**lift-definition** *less-eq-unit* :: *unit* ⇒ *unit* ⇒ *bool* **is** *less-eq* ⟨*proof*⟩

**lift-definition** *less-unit* :: *unit* ⇒ *unit* ⇒ *bool* **is** *less* ⟨*proof*⟩

**instance**
⟨*proof*⟩

**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* ≡ *max* (*x* + *y* − *1*) *0*

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

**lemma** *tl-top-zero*:
$x \in \{0..1\} \Longrightarrow tl\ 1\ x = x$
⟨*proof*⟩

**lift-definition** *plus-unit* :: *unit* ⇒ *unit* ⇒ *unit* **is** *tl*
⟨*proof*⟩

**instance**
⟨*proof*⟩

**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* ⟨*proof*⟩

**subclass** *aggregation-order*
⟨*proof*⟩

**sublocale** *linear-aggregation-kleene-algebra* **where** *sup* = *max* **and** *inf* = *min* **and** *times* = *min* **and** *conv* = *id* **and** *one* = *top* **and** *star* = λ*x* . *top* **and** *uminus* = λ*x* . *if x* = *bot then top else bot*
⟨*proof*⟩

**lemma** *t-top*: *x* + *top* = *top*
⟨*proof*⟩

**lemma** *add-increasing*: *x* ≤ *x* + *y*
⟨*proof*⟩

**lemma** *t-max*: *max x y* ≤ *x* + *y*
⟨*proof*⟩

**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*
⟨*proof*⟩

**setup-lifting** *type-definition-unit2*

**instantiation** *unit2* :: *bounded-linorder*
**begin**

**lift-definition** *bot-unit2* :: *unit2* **is** *0*
⟨*proof*⟩

**lift-definition** *top-unit2* :: *unit2* **is** *1*
⟨*proof*⟩

**lift-definition** *less-eq-unit2* :: *unit2* ⇒ *unit2* ⇒ *bool* **is** *less-eq* ⟨*proof*⟩

**lift-definition** *less-unit2* :: *unit2* ⇒ *unit2* ⇒ *bool* **is** *less* ⟨*proof*⟩

**instance**
⟨*proof*⟩

**end**

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

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

**abbreviation** *sp* :: *real* ⇒ *real* ⇒ *real* **where**
*sp x y* ≡ *x* + *y* − *x* ∗ *y*

**lemma** *sp-assoc*:
*sp* (*sp x y*) *z* = *sp x* (*sp y z*)
⟨*proof*⟩

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

**lift-definition** *plus-unit2* :: *unit2* ⇒ *unit2* ⇒ *unit2* **is** *sp*
⟨*proof*⟩

**instance**
⟨*proof*⟩

**end**

## 5.6 Constant Aggregation

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

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

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

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

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

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

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

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

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

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

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

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

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

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

**instance**
⟨*proof*⟩

**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** ⟨*proof*⟩

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

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

## 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* ≤ *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* ≤ *y* ∧ ¬ *y* ≤ *x*)

**instance**
⟨*proof*⟩

**end**

**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.