# Kleene Algebra

Alasdair Armstrong, Victor B. F. Gomes, Georg Struth and Tjark Weber

October 10, 2017

**Abstract**

Variants of Dioids and Kleene algebras are formalised together with their most important models in Isabelle/HOL. The Kleene algebras presented include process algebras based on bisimulation equivalence (near Kleene algebras), simulation equivalence (pre-Kleene algebras) and language equivalence (Kleene algebras), as well as algebras with ambiguous finite or infinite iteration (Conway algebras), possibly infinite iteration (demonic refinement algebras), infinite iteration (omega algebras) and residuated variants (action algebras). Models implemented include binary relations, (regular) languages, sets of paths and traces, power series and matrices. Finally, min-plus and max-plus algebras as well as generalised Hoare logics for Kleene algebras and demonic refinement algebras are provided for applications.

## Contents

# 1 Introductory Remarks

These theory files are intended as a reference formalisation of variants of Kleene algebras and as a basis for other variants, such as Kleene algebras with tests [2] and modal Kleene algebras [14], which are useful for program correctness and verification. To that end we have aimed at making proof accessible to readers at textbook granularity instead of fully automating them. In that sense, these files can be considered a machine-checked introduction to reasoning in Kleene algebra.

Beyond that, the theories are only sparsely commented. Additional information on the hierarchy of Kleene algebras and its formalisation in Isabelle/HOL can be found in a tutorial paper [13] or an overview article [17]. While these papers focus on the automation of algebraic reasoning, the present formalisation presents readable proofs whenever these are interesting and instructive.

Expansions of the hierarchy to modal Kleene algebras, Kleene algebras with tests and Hoare logics as well as infinitary and higher-order Kleene alge-

bras [16, 3], and an alternative hierarchy of regular algebras and Kleene algebras [11]—orthogonal to the present one—have also been implemented in the Archive of Formal Proofs [12, 14, 2, 1].

# 2 Signatures

**theory** *Signatures*
**imports** *Main*
**begin**

Default notation in Isabelle/HOL is occasionally different from established notation in the relation/algebra community. We use the latter where possible.

**notation**
  *times* (**infixl** · *70*)

Some classes in our algebraic hierarchy are most naturally defined as subclasses of two (or more) superclasses that impose different restrictions on the same parameter(s).

Alas, in Isabelle/HOL, a class cannot have multiple superclasses that independently declare the same parameter(s). One workaround, which motivated the following syntactic classes, is to shift the parameter declaration to a common superclass.

**class** *star-op* =
  **fixes** *star* :: $'a \Rightarrow 'a$ ($-^\star$ [*101*] *100*)

**class** *omega-op* =
  **fixes** *omega* :: $'a \Rightarrow 'a$ ($-^\omega$ [*101*] *100*)

We define a type class that combines addition and the definition of order in, e.g., semilattices. This class makes the definition of various other type classes more slick.

**class** *plus-ord* = *plus* + *ord* +
  **assumes** *less-eq-def*: $x \leq y \longleftrightarrow x + y = y$
  **and** *less-def*: $x < y \longleftrightarrow x \leq y \wedge x \neq y$

**end**

# 3 Dioids

**theory** *Dioid*
**imports** *Signatures*
**begin**

## 3.1 Join Semilattices

Join semilattices can be axiomatised order-theoretically or algebraically. A join semilattice (or upper semilattice) is either a poset in which every pair of elements has a join (or least upper bound), or a set endowed with an associative, commutative, idempotent binary operation. It is well known that the order-theoretic definition induces the algebraic one and vice versa. We start from the algebraic axiomatisation because it is easily expandable to dioids, using Isabelle's type class mechanism.

In Isabelle/HOL, a type class *semilattice-sup* is available. Alas, we cannot use this type class because we need the symbol + for the join operation in the dioid expansion and subclass proofs in Isabelle/HOL require the two type classes involved to have the same fixed signature.

Using *add_assoc* as a name for the first assumption in class *join_semilattice* would lead to name clashes: we will later define classes that inherit from *semigroup-add*, which provides its own assumption *add_assoc*, and prove that these are subclasses of *join_semilattice*. Hence the primed name.

**class** *join-semilattice* = *plus-ord* +
  **assumes** *add-assoc'* [*ac-simps*]: $(x + y) + z = x + (y + z)$
  **and** *add-comm* [*ac-simps*] : $x + y = y + x$
  **and** *add-idem* [*simp*]: $x + x = x$
**begin**

**lemma** *add-left-comm* [*ac-simps*]: $y + (x + z) = x + (y + z)$
  ⟨*proof*⟩

**lemma** *add-left-idem* [*ac-simps*]: $x + (x + y) = x + y$
  ⟨*proof*⟩

The definition $(x \leq y) = (x + y = y)$ of the order is hidden in class *plus-ord*. We show some simple order-based properties of semilattices. The first one states that every semilattice is a partial order.

**subclass** *order*
⟨*proof*⟩

Next we show that joins are least upper bounds.

**sublocale** *join*: *semilattice-sup op* +
  ⟨*proof*⟩

Next we prove that joins are isotone (order preserving).

**lemma** *add-iso*: $x \leq y \Longrightarrow x + z \leq y + z$
  ⟨*proof*⟩

The next lemma links the definition of order as $(x \leq y) = (x + y = y)$ with a perhaps more conventional one known, e.g., from arithmetics.

**lemma** *order-prop*: $x \leq y \longleftrightarrow (\exists\, z.\ x + z = y)$
⟨*proof*⟩

**end**

## 3.2 Join Semilattices with an Additive Unit

We now expand join semilattices by an additive unit 0. Is the least element with respect to the order, and therefore often denoted by $\bot$. Semilattices with a least element are often called *bounded*.

**class** *join-semilattice-zero* = *join-semilattice* + *zero* +
  **assumes** *add-zero-l* [*simp*]: $0 + x = x$

**begin**

**subclass** *comm-monoid-add*
  ⟨*proof*⟩

**sublocale** *join*: *bounded-semilattice-sup-bot op* + *op* $\leq$ *op* $<$ *0*
  ⟨*proof*⟩

**lemma** *no-trivial-inverse*: $x \neq 0 \Longrightarrow \neg(\exists\, y.\ x + y = 0)$
  ⟨*proof*⟩

**end**

## 3.3 Near Semirings

*Near semirings* (also called seminearrings) are generalisations of near rings to the semiring case. They have been studied, for instance, in G. Pilz's book [25] on near rings. According to his definition, a near semiring consists of an additive and a multiplicative semigroup that interact via a single distributivity law (left or right). The additive semigroup is not required to be commutative. The definition is influenced by partial transformation semigroups.

We only consider near semirings in which addition is commutative, and in which the right distributivity law holds. We call such near semirings *abelian*.

**class** *ab-near-semiring* = *ab-semigroup-add* + *semigroup-mult* +
  **assumes** *distrib-right'* [*simp*]: $(x + y) \cdot z = x \cdot z + y \cdot z$

**subclass** (**in** *semiring*) *ab-near-semiring*
  ⟨*proof*⟩

**class** *ab-pre-semiring* = *ab-near-semiring* +
  **assumes** *subdistl-eq*: $z \cdot x + z \cdot (x + y) = z \cdot (x + y)$

### 3.4 Variants of Dioids

A *near dioid* is an abelian near semiring in which addition is idempotent. This generalises the notion of (additively) idempotent semirings by dropping one distributivity law. Near dioids are a starting point for process algebras.

By modelling variants of dioids as variants of semirings in which addition is idempotent we follow the tradition of Birkhoff [5], but deviate from the definitions in Gondran and Minoux's book [15].

**class** *near-dioid* = *ab-near-semiring* + *plus-ord* +
  **assumes** *add-idem'* [*simp*]: $x + x = x$

**begin**

Since addition is idempotent, the additive (commutative) semigroup reduct of a near dioid is a semilattice. Near dioids are therefore ordered by the semilattice order.

**subclass** *join-semilattice*
  ⟨*proof*⟩

It follows that multiplication is right-isotone (but not necessarily left-isotone).

**lemma** *mult-isor*: $x \leq y \Longrightarrow x \cdot z \leq y \cdot z$
⟨*proof*⟩

**lemma** $x \leq y \Longrightarrow z \cdot x \leq z \cdot y$

⟨*proof*⟩

The next lemma states that, in every near dioid, left isotonicity and left subdistributivity are equivalent.

**lemma** *mult-isol-equiv-subdistl*:
  $(\forall\, x\ y\ z.\ x \leq y \longrightarrow z \cdot x \leq z \cdot y) \longleftrightarrow (\forall\, x\ y\ z.\ z \cdot x \leq z \cdot (x + y))$
  ⟨*proof*⟩

The following lemma is relevant to propositional Hoare logic.

**lemma** *phl-cons1*: $x \leq w \Longrightarrow w \cdot y \leq y \cdot z \Longrightarrow x \cdot y \leq y \cdot z$
  ⟨*proof*⟩

**end**

We now make multiplication in near dioids left isotone, which is equivalent to left subdistributivity, as we have seen. The corresponding structures form the basis of probabilistic Kleene algebras [24] and game algebras [29]. We are not aware that these structures have a special name, so we baptise them *pre-dioids*.

We do not explicitly define pre-semirings since we have no application for them.

**class** *pre-dioid* = *near-dioid* +
  **assumes** *subdistl*: $z \cdot x \leq z \cdot (x + y)$

**begin**

Now, obviously, left isotonicity follows from left subdistributivity.

**lemma** *subdistl-var*: $z \cdot x + z \cdot y \leq z \cdot (x + y)$
  ⟨*proof*⟩

**subclass** *ab-pre-semiring*
  ⟨*proof*⟩

**lemma** *mult-isol*: $x \leq y \Longrightarrow z \cdot x \leq z \cdot y$
⟨*proof*⟩

**lemma** *mult-isol-var*: $u \leq x \Longrightarrow v \leq y \Longrightarrow u \cdot v \leq x \cdot y$
  ⟨*proof*⟩

**lemma** *mult-double-iso*: $x \leq y \Longrightarrow w \cdot x \cdot z \leq w \cdot y \cdot z$
  ⟨*proof*⟩

The following lemmas are relevant to propositional Hoare logic.

**lemma** *phl-cons2*: $w \leq x \Longrightarrow z \cdot y \leq y \cdot w \Longrightarrow z \cdot y \leq y \cdot x$
  ⟨*proof*⟩

**lemma** *phl-seq*:
**assumes** $p \cdot x \leq x \cdot r$
**and** $r \cdot y \leq y \cdot q$
**shows** $p \cdot (x \cdot y) \leq x \cdot y \cdot q$
⟨*proof*⟩

**lemma** *phl-cond*:
**assumes** $u \cdot v \leq v \cdot u \cdot v$ **and** $u \cdot w \leq w \cdot u \cdot w$
**and** $\bigwedge x\ y.\ u \cdot (x + y) \leq u \cdot x + u \cdot y$
**and** $u \cdot v \cdot x \leq x \cdot z$ **and** $u \cdot w \cdot y \leq y \cdot z$
**shows** $u \cdot (v \cdot x + w \cdot y) \leq (v \cdot x + w \cdot y) \cdot z$
⟨*proof*⟩

**lemma** *phl-export1*:
**assumes** $x \cdot y \leq y \cdot x \cdot y$
**and** $(x \cdot y) \cdot z \leq z \cdot w$
**shows** $x \cdot (y \cdot z) \leq (y \cdot z) \cdot w$
⟨*proof*⟩

**lemma** *phl-export2*:
**assumes** $z \cdot w \leq w \cdot z \cdot w$
**and** $x \cdot y \leq y \cdot z$
**shows** $x \cdot (y \cdot w) \leq y \cdot w \cdot (z \cdot w)$
⟨*proof*⟩

**end**

By adding a full left distributivity law we obtain semirings (which are already available in Isabelle/HOL as *semiring*) from near semirings, and dioids from near dioids. Dioids are therefore idempotent semirings.

**class** *dioid* = *near-dioid* + *semiring*

**subclass** (**in** *dioid*) *pre-dioid*
⟨*proof*⟩

## 3.5 Families of Nearsemirings with a Multiplicative Unit

Multiplicative units are important, for instance, for defining an operation of finite iteration or Kleene star on dioids. We do not introduce left and right units separately since we have no application for this.

**class** *ab-near-semiring-one* = *ab-near-semiring* + *one* +
**assumes** *mult-onel* [*simp*]: $1 \cdot x = x$
**and** *mult-oner* [*simp*]: $x \cdot 1 = x$

**begin**

**subclass** *monoid-mult*
⟨*proof*⟩

**end**

**class** *ab-pre-semiring-one* = *ab-near-semiring-one* + *ab-pre-semiring*

**class** *near-dioid-one* = *near-dioid* + *ab-near-semiring-one*

**begin**

The following lemma is relevant to propositional Hoare logic.

**lemma** *phl-skip*: $x \cdot 1 \leq 1 \cdot x$
⟨*proof*⟩

**end**

For near dioids with one, it would be sufficient to require $1 + 1 = 1$. This implies $x + x = x$ for arbitray $x$ (but that would lead to annoying redundant proof obligations in mutual subclasses of *near-dioid-one* and *near-dioid* later).

**class** *pre-dioid-one* = *pre-dioid* + *near-dioid-one*

**class** *dioid-one* = *dioid* + *near-dioid-one*

**subclass** (**in** *dioid-one*) *pre-dioid-one* ⟨*proof*⟩

### 3.6 Families of Nearsemirings with Additive Units

We now axiomatise an additive unit 0 for nearsemirings. The zero is usually required to satisfy annihilation properties with respect to multiplication. Due to applications we distinguish a zero which is only a left annihilator from one that is also a right annihilator. More briefly, we call zero either a left unit or a unit.

Semirings and dioids with a right zero only can be obtained from those with a left unit by duality.

**class** *ab-near-semiring-one-zerol* = *ab-near-semiring-one* + *zero* +
  **assumes** *add-zerol* [*simp*]: *0* + *x* = *x*
  **and** *annil* [*simp*]: *0* · *x* = *0*

**begin**

Note that we do not require $0 \neq 1$.

**lemma** *add-zeror* [*simp*]: *x* + *0* = *x*
  ⟨*proof*⟩

**end**

**class** *ab-pre-semiring-one-zerol* = *ab-near-semiring-one-zerol* + *ab-pre-semiring*

**begin**

The following lemma shows that there is no point defining pre-semirings separately from dioids.

**lemma** *1* + *1* = *1*
⟨*proof*⟩

**end**

**class** *near-dioid-one-zerol* = *near-dioid-one* + *ab-near-semiring-one-zerol*

**subclass** (**in** *near-dioid-one-zerol*) *join-semilattice-zero*
  ⟨*proof*⟩

**class** *pre-dioid-one-zerol* = *pre-dioid-one* + *ab-near-semiring-one-zerol*

**subclass** (**in** *pre-dioid-one-zerol*) *near-dioid-one-zerol* ⟨*proof*⟩

**class** *semiring-one-zerol* = *semiring* + *ab-near-semiring-one-zerol*

**class** *dioid-one-zerol* = *dioid-one* + *ab-near-semiring-one-zerol*

**subclass** (**in** *dioid-one-zerol*) *pre-dioid-one-zerol* ⟨*proof*⟩

We now make zero also a right annihilator.

**class** *ab-near-semiring-one-zero* = *ab-near-semiring-one-zerol* +
  **assumes** *annir* [*simp*]: $x \cdot 0 = 0$

**class** *semiring-one-zero* = *semiring* + *ab-near-semiring-one-zero*

**class** *near-dioid-one-zero* = *near-dioid-one-zerol* + *ab-near-semiring-one-zero*

**class** *pre-dioid-one-zero* = *pre-dioid-one-zerol* + *ab-near-semiring-one-zero*

**subclass** (**in** *pre-dioid-one-zero*) *near-dioid-one-zero* ⟨*proof*⟩

**class** *dioid-one-zero* = *dioid-one-zerol* + *ab-near-semiring-one-zero*

**subclass** (**in** *dioid-one-zero*) *pre-dioid-one-zero* ⟨*proof*⟩

**subclass** (**in** *dioid-one-zero*) *semiring-one-zero* ⟨*proof*⟩

## 3.7 Duality by Opposition

Swapping the order of multiplication in a semiring (or dioid) gives another semiring (or dioid), called its *dual* or *opposite*.

**definition** (**in** *times*) *opp-mult* (**infixl** $\odot$ *70*)
  **where** $x \odot y \equiv y \cdot x$

**lemma** (**in** *semiring-1*) *dual-semiring-1*:
  *class.semiring-1 1* (*op* $\odot$) (*op* +) *0*
  ⟨*proof*⟩

**lemma** (**in** *dioid-one-zero*) *dual-dioid-one-zero*:
  *class.dioid-one-zero* (*op* +) (*op* $\odot$) *1 0* (*op* $\leq$) (*op* <)
  ⟨*proof*⟩

## 3.8 Selective Near Semirings

In this section we briefly sketch a generalisation of the notion of *dioid*. Some important models, e.g. max-plus and min-plus semirings, have that property.

**class** *selective-near-semiring* = *ab-near-semiring* + *plus-ord* +
  **assumes** *select*: $x + y = x \vee x + y = y$

**begin**

**lemma** *select-alt*: $x + y \in \{x,y\}$
  ⟨*proof*⟩

It follows immediately that every selective near semiring is a near dioid.

**subclass** *near-dioid*
  ⟨*proof*⟩

Moreover, the order in a selective near semiring is obviously linear.

**subclass** *linorder*
⟨*proof*⟩

**end**

**class** *selective-semiring* = *selective-near-semiring* + *semiring-one-zero*

**begin**

**subclass** *dioid-one-zero* ⟨*proof*⟩

**end**

**end**

# 4 Models of Dioids

**theory** *Dioid-Models*
**imports** *Dioid HOL.Real*
**begin**

In this section we consider some well known models of dioids. These so far include the powerset dioid over a monoid, languages, binary relations, sets of traces, sets paths (in a graph), as well as the min-plus and the max-plus semirings. Most of these models are taken from an article about Kleene algebras with domain [9].

The advantage of formally linking these models with the abstract axiomatisations of dioids is that all abstract theorems are automatically available in all models. It therefore makes sense to establish models for the strongest possible axiomatisations (whereas theorems should be proved for the weakest ones).

## 4.1 The Powerset Dioid over a Monoid

We assume a multiplicative monoid and define the usual complex product on sets of elements. We formalise the well known result that this lifting induces a dioid.

**instantiation** *set* :: (*monoid-mult*) *monoid-mult*
**begin**

**definition** *one-set-def*:
$1 = \{1\}$

**definition** *c-prod-def*: — the complex product
$A \cdot B = \{u * v \mid u\ v.\ u \in A \wedge v \in B\}$

**instance**
⟨*proof*⟩

**end**

**instantiation** *set* :: (*monoid-mult*) *dioid-one-zero*
**begin**

**definition** *zero-set-def*:
*0* = {}

**definition** *plus-set-def*:
*A* + *B* = *A* ∪ *B*

**instance**
⟨*proof*⟩

**end**

## 4.2 Language Dioids

Language dioids arise as special cases of the monoidal lifting because sets of words form free monoids. Moreover, monoids of words are isomorphic to monoids of lists under append.

To show that languages form dioids it therefore suffices to show that sets of lists closed under append and multiplication with the empty word form a (multiplicative) monoid. Isabelle then does the rest of the work automatically. Infix @ denotes word concatenation.

**instantiation** *list* :: (*type*) *monoid-mult*
**begin**

**definition** *times-list-def*:
*xs* ∗ *ys* ≡ *xs* @ *ys*

**definition** *one-list-def*:
*1* ≡ []

**instance** ⟨*proof*⟩

**end**

Languages as sets of lists have already been formalised in Isabelle in various places. We can now obtain much of their algebra for free.

**type-synonym** *'a lan* = *'a list set*

**interpretation** *lan-dioid*: *dioid-one-zero op* + *op* · *1*::*'a lan 0 op* ⊆ *op* ⊂ ⟨*proof*⟩

## 4.3 Relation Dioids

We now show that binary relations under union, relational composition, the identity relation, the empty relation and set inclusion form dioids. Due to the well developed relation library of Isabelle this is entirely trivial.

**interpretation** *rel-dioid*: *dioid-one-zero op* $\cup$ *op O Id* $\{\}$ *op* $\subseteq$ *op* $\subset$
⟨*proof*⟩

**interpretation** *rel-monoid*: *monoid-mult Id op O* ⟨*proof*⟩

## 4.4 Trace Dioids

Traces have been considered, for instance, by Kozen [22] in the context of Kleene algebras with tests. Intuitively, a trace is an execution sequence of a labelled transition system from some state to some other state, in which state labels and action labels alternate, and which begin and end with a state label.

Traces generalise words: words can be obtained from traces by forgetting state labels. Similarly, sets of traces generalise languages.

In this section we show that sets of traces under union, an appropriately defined notion of complex product, the set of all traces of length zero, the empty set of traces and set inclusion form a dioid.

We first define the notion of trace and the product of traces, which has been called *fusion product* by Kozen.

**type-synonym** $('p, 'a)$ *trace* $= 'p \times ('a \times 'p)$ *list*

**definition** *first* :: $('p, 'a)$ *trace* $\Rightarrow 'p$ **where**
*first* = *fst*

**lemma** *first-conv* [*simp*]: *first* $(p, xs) = p$
⟨*proof*⟩

**fun** *last* :: $('p, 'a)$ *trace* $\Rightarrow 'p$ **where**
*last* $(p, [])$ = $p$
| *last* $(\text{-}, xs)$ = *snd* (*List.last xs*)

**lemma** *last-append* [*simp*]: *last* $(p, xs$ @ $ys)$ = *last* $($*last* $(p, xs), ys)$
⟨*proof*⟩

The fusion product is a partial operation. It is undefined if the last element of the first trace and the first element of the second trace are different. If these elements are the same, then the fusion product removes the first element from the second trace and appends the resulting object to the first trace.

**definition** *t-fusion* :: $('p, 'a)$ *trace* $\Rightarrow$ $('p, 'a)$ *trace* $\Rightarrow$ $('p, 'a)$ *trace* **where**

*t-fusion x y ≡ if last x = first y then (fst x, snd x @ snd y) else undefined*

We now show that the first element and the last element of a trace are a left and right unit for that trace and prove some other auxiliary lemmas.

**lemma** *t-fusion-leftneutral* [*simp*]: *t-fusion* (*first x*, []) *x* = *x*
⟨*proof*⟩

**lemma** *fusion-rightneutral* [*simp*]: *t-fusion x* (*last x*, []) = *x*
⟨*proof*⟩

**lemma** *first-t-fusion* [*simp*]: *last x* = *first y* ⟹ *first* (*t-fusion x y*) = *first x*
⟨*proof*⟩

**lemma** *last-t-fusion* [*simp*]: *last x* = *first y* ⟹ *last* (*t-fusion x y*) = *last y*
⟨*proof*⟩

Next we show that fusion of traces is associative.

**lemma** *t-fusion-assoc* [*simp*]:
⟦ *last x* = *first y*; *last y* = *first z* ⟧ ⟹ *t-fusion x* (*t-fusion y z*) = *t-fusion* (*t-fusion x y*) *z*
⟨*proof*⟩

## 4.5 Sets of Traces

We now lift the fusion product to a complex product on sets of traces. This operation is total.

**no-notation**
*times* (**infixl** · *70*)

**definition** *t-prod* :: ('*p*, '*a*) *trace set* ⇒ ('*p*, '*a*) *trace set* ⇒ ('*p*, '*a*) *trace set* (**infixl** · *70*)
**where** *X* · *Y* = {*t-fusion u v*| *u v*. *u* ∈ *X* ∧ *v* ∈ *Y* ∧ *last u* = *first v*}

Next we define the empty set of traces and the set of traces of length zero as the multiplicative unit of the trace dioid.

**definition** *t-zero* :: ('*p*, '*a*) *trace set* **where**
*t-zero* ≡ {}

**definition** *t-one* :: ('*p*, '*a*) *trace set* **where**
*t-one* ≡ ⋃ *p*. {(*p*, [])}

We now provide elimination rules for trace products.

**lemma** *t-prod-iff*:
*w* ∈ *X*·*Y* ⟷ (∃ *u v*. *w* = *t-fusion u v* ∧ *u* ∈ *X* ∧ *v* ∈ *Y* ∧ *last u* = *first v*)
⟨*proof*⟩

**lemma** *t-prod-intro* [*simp*, *intro*]:

$[\![ u \in X;\ v \in Y;\ last\ u = first\ v ]\!] \Longrightarrow t\text{-}fusion\ u\ v \in X \cdot Y$
⟨*proof*⟩

**lemma** *t-prod-elim* [*elim*]:
$w \in X \cdot Y \Longrightarrow \exists\, u\ v.\ w = t\text{-}fusion\ u\ v \wedge u \in X \wedge v \in Y \wedge last\ u = first\ v$
⟨*proof*⟩

Finally we prove the interpretation statement that sets of traces under union and the complex product based on trace fusion together with the empty set of traces and the set of traces of length one forms a dioid.

**interpretation** *trace-dioid*: *dioid-one-zero op* $\cup$ *t-prod t-one t-zero op* $\subseteq$ *op* $\subset$
⟨*proof*⟩

**no-notation**
*t-prod* (**infixl** $\cdot$ *70*)

## 4.6 The Path Diod

The next model we consider are sets of paths in a graph. We consider two variants, one that contains the empty path and one that doesn't. The former leads to more difficult proofs and a more involved specification of the complex product. We start with paths that include the empty path. In this setting, a path is a list of nodes.

## 4.7 Path Models with the Empty Path

**type-synonym** *'a path* = *'a list*

Path fusion is defined similarly to trace fusion. Mathematically it should be a partial operation. The fusion of two empty paths yields the empty path; the fusion between a non-empty path and an empty one is undefined; the fusion of two non-empty paths appends the tail of the second path to the first one.

We need to use a total alternative and make sure that undefined paths do not contribute to the complex product.

**fun** *p-fusion* :: *'a path* ⇒ *'a path* ⇒ *'a path* **where**
*p-fusion* [] - = []
| *p-fusion* - [] = []
| *p-fusion ps* (*q* # *qs*) = *ps* @ *qs*

**lemma** *p-fusion-assoc*:
*p-fusion ps* (*p-fusion qs rs*) = *p-fusion* (*p-fusion ps qs*) *rs*
⟨*proof*⟩

This lemma overapproximates the real situation, but it holds in all cases where path fusion should be defined.

**lemma** *p-fusion-last*:
**assumes** *List.last ps* = *hd qs*
**and** *ps* ≠ []
**and** *qs* ≠ []
**shows** *List.last* (*p-fusion ps qs*) = *List.last qs*
⟨*proof*⟩

**lemma** *p-fusion-hd*: ⟦*ps* ≠ []; *qs* ≠ []⟧ ⟹ *hd* (*p-fusion ps qs*) = *hd ps*
⟨*proof*⟩

**lemma** *nonempty-p-fusion*: ⟦*ps* ≠ []; *qs* ≠ []⟧ ⟹ *p-fusion ps qs* ≠ []
⟨*proof*⟩

We now define a condition that filters out undefined paths in the complex product.

**abbreviation** *p-filter* :: *'a path* ⇒ *'a path* ⇒ *bool* **where**
*p-filter ps qs* ≡ ((*ps* = [] ∧ *qs* = []) ∨ (*ps* ≠ [] ∧ *qs* ≠ [] ∧ (*List.last ps*) = *hd qs*))

**no-notation**
*times* (**infixl** · *70*)

**definition** *p-prod* :: *'a path set* ⇒ *'a path set* ⇒ *'a path set* (**infixl** · *70*)
**where** *X* · *Y* = {*rs* . ∃ *ps* ∈ *X*. ∃ *qs* ∈ *Y*. *rs* = *p-fusion ps qs* ∧ *p-filter ps qs*}

**lemma** *p-prod-iff*:
*ps* ∈ *X* · *Y* ⟷ (∃ *qs rs*. *ps* = *p-fusion qs rs* ∧ *qs* ∈ *X* ∧ *rs* ∈ *Y* ∧ *p-filter qs rs*)
⟨*proof*⟩

Due to the complexity of the filter condition, proving properties of complex products can be tedious.

**lemma** *p-prod-assoc*: (*X* · *Y*) · *Z* = *X* · (*Y* · *Z*)
⟨*proof*⟩

We now define the multiplicative unit of the path dioid as the set of all paths of length one, including the empty path, and show the unit laws with respect to the path product.

**definition** *p-one* :: *'a path set* **where**
*p-one* ≡ {*p* . ∃ *q*::*'a*. *p* = [*q*]} ∪ {[]}

**lemma** *p-prod-onel* [*simp*]: *p-one* · *X* = *X*
⟨*proof*⟩

**lemma** *p-prod-oner* [*simp*]: *X* · *p-one* = *X*
⟨*proof*⟩

Next we show distributivity laws at the powerset level.

**lemma** *p-prod-distl*: *X* · (*Y* ∪ *Z*) = *X* · *Y* ∪ *X* · *Z*

⟨*proof*⟩

**lemma** *p-prod-distr*: $(X \cup Y) \cdot Z = X \cdot Z \cup Y \cdot Z$
⟨*proof*⟩

Finally we show that sets of paths under union, the complex product, the unit set and the empty set form a dioid.

**interpretation** *path-dioid*: *dioid-one-zero op* $\cup$ *op* $\cdot$ *p-one* $\{\}$ *op* $\subseteq$ *op* $\subset$
⟨*proof*⟩

**no-notation**
*p-prod* (**infixl** $\cdot$ *70*)

## 4.8 Path Models without the Empty Path

We now build a model of paths that does not include the empty path and therefore leads to a simpler complex product.

**datatype** *'a ppath* = *Node 'a* | *Cons 'a 'a ppath*

**primrec** *pp-first* :: *'a ppath* ⇒ *'a* **where**
*pp-first* (*Node x*) = *x*
| *pp-first* (*Cons x* -) = *x*

**primrec** *pp-last* :: *'a ppath* ⇒ *'a* **where**
*pp-last* (*Node x*) = *x*
| *pp-last* (*Cons* - *xs*) = *pp-last xs*

The path fusion product (although we define it as a total funcion) should only be applied when the last element of the first argument is equal to the first element of the second argument.

**primrec** *pp-fusion* :: *'a ppath* ⇒ *'a ppath* ⇒ *'a ppath* **where**
*pp-fusion* (*Node x*) *ys* = *ys*
| *pp-fusion* (*Cons x xs*) *ys* = *Cons x* (*pp-fusion xs ys*)

We now go through the same steps as for traces and paths before, showing that the first and last element of a trace a left or right unit for that trace and that the fusion product on traces is associative.

**lemma** *pp-fusion-leftneutral* [*simp*]: *pp-fusion* (*Node* (*pp-first x*)) *x* = *x*
⟨*proof*⟩

**lemma** *pp-fusion-rightneutral* [*simp*]: *pp-fusion x* (*Node* (*pp-last x*)) = *x*
⟨*proof*⟩

**lemma** *pp-first-pp-fusion* [*simp*]:
*pp-last x* = *pp-first y* ⟹ *pp-first* (*pp-fusion x y*) = *pp-first x*
⟨*proof*⟩

**lemma** *pp-last-pp-fusion* [*simp*]:
*pp-last x* = *pp-first y* ⟹ *pp-last* (*pp-fusion x y*) = *pp-last y*
⟨*proof*⟩

**lemma** *pp-fusion-assoc* [*simp*]:
⟦ *pp-last x* = *pp-first y*; *pp-last y* = *pp-first z* ⟧ ⟹ *pp-fusion x* (*pp-fusion y z*) = *pp-fusion* (*pp-fusion x y*) *z*
⟨*proof*⟩

We now lift the path fusion product to a complex product on sets of paths. This operation is total.

**definition** *pp-prod* :: *'a ppath set* ⇒ *'a ppath set* ⇒ *'a ppath set* (**infixl** · *70*)
**where** *X*·*Y* = {*pp-fusion u v*| *u v*. *u* ∈ *X* ∧ *v* ∈ *Y* ∧ *pp-last u* = *pp-first v*}

Next we define the set of paths of length one as the multiplicative unit of the path dioid.

**definition** *pp-one* :: *'a ppath set* **where**
*pp-one* ≡ *range Node*

We again provide an elimination rule.

**lemma** *pp-prod-iff*:
*w* ∈ *X*·*Y* ⟷ (∃ *u v*. *w* = *pp-fusion u v* ∧ *u* ∈ *X* ∧ *v* ∈ *Y* ∧ *pp-last u* = *pp-first v*)
⟨*proof*⟩

**interpretation** *ppath-dioid*: *dioid-one-zero op* ∪ *op* · *pp-one* {} *op* ⊆ *op* ⊂
⟨*proof*⟩

**no-notation**
*pp-prod* (**infixl** · *70*)

## 4.9 The Distributive Lattice Dioid

A bounded distributive lattice is a distributive lattice with a least and a greatest element. Using Isabelle's lattice theory file we define a bounded distributive lattice as an axiomatic type class and show, using a sublocale statement, that every bounded distributive lattice is a dioid with one and zero.

**class** *bounded-distributive-lattice* = *bounded-lattice* + *distrib-lattice*

**sublocale** *bounded-distributive-lattice* ⊆ *dioid-one-zero sup inf top bot less-eq*
⟨*proof*⟩

## 4.10 The Boolean Dioid

In this section we show that the booleans form a dioid, because the booleans form a bounded distributive lattice.

**instantiation** *bool* :: *bounded-distributive-lattice*
**begin**

**instance** ⟨*proof*⟩

**end**

**interpretation** *boolean-dioid*: *dioid-one-zero sup inf True False less-eq less*
⟨*proof*⟩

## 4.11 The Max-Plus Dioid

The following dioids have important applications in combinatorial optimisations, control theory, algorithm design and computer networks.

A definition of reals extended with $+\infty$ *and* $-\infty$ may be found in *HOL/Library/Extended_Real.thy*. Alas, we require separate extensions with either $+\infty$ or $-\infty$.

The carrier set of the max-plus semiring is the set of real numbers extended by minus infinity. The operation of addition is maximum, the operation of multiplication is addition, the additive unit is minus infinity and the multiplicative unit is zero.

**datatype** *mreal* = *mreal real* | *MInfty* — minus infinity

**fun** *mreal-max* **where**
*mreal-max* (*mreal x*) (*mreal y*) = *mreal* (*max x y*)
| *mreal-max x MInfty* = *x*
| *mreal-max MInfty y* = *y*

**lemma** *mreal-max-simp-3* [*simp*]: *mreal-max MInfty y* = *y*
⟨*proof*⟩

**fun** *mreal-plus* **where**
*mreal-plus* (*mreal x*) (*mreal y*) = *mreal* (*x* + *y*)
| *mreal-plus* - - = *MInfty*

We now show that the max plus-semiring satisfies the axioms of selective semirings, from which it follows that it satisfies the dioid axioms.

**instantiation** *mreal* :: *selective-semiring*
**begin**

**definition** *zero-mreal-def*:
*0* ≡ *MInfty*

**definition** *one-mreal-def*:
*1* ≡ *mreal 0*

**definition** *plus-mreal-def*:

$x + y \equiv mreal\text{-}max\ x\ y$

**definition** *times-mreal-def*:
$x * y \equiv mreal\text{-}plus\ x\ y$

**definition** *less-eq-mreal-def*:
$(x::mreal) \leq y \equiv x + y = y$

**definition** *less-mreal-def*:
$(x::mreal) < y \equiv x \leq y \wedge x \neq y$

**instance**
⟨*proof*⟩

**end**

## 4.12 The Min-Plus Dioid

The min-plus dioid is also known as *tropical semiring*. Here we need to add a positive infinity to the real numbers. The procedere follows that of max-plus semirings.

**datatype** *preal* = *preal real* | *PInfty* — plus infinity

**fun** *preal-min* **where**
*preal-min* (*preal x*) (*preal y*) = *preal* (*min x y*)
| *preal-min x PInfty* = *x*
| *preal-min PInfty y* = *y*

**lemma** *preal-min-simp-3* [*simp*]: *preal-min PInfty y* = *y*
⟨*proof*⟩

**fun** *preal-plus* **where**
*preal-plus* (*preal x*) (*preal y*) = *preal* (*x* + *y*)
| *preal-plus* - - = *PInfty*

**instantiation** *preal* :: *selective-semiring*
**begin**

**definition** *zero-preal-def*:
*0* ≡ *PInfty*

**definition** *one-preal-def*:
*1* ≡ *preal 0*

**definition** *plus-preal-def*:
$x + y \equiv preal\text{-}min\ x\ y$

**definition** *times-preal-def*:
$x * y \equiv preal\text{-}plus\ x\ y$

**definition** *less-eq-preal-def*:
$(x::preal) \leq y \equiv x + y = y$

**definition** *less-preal-def*:
$(x::preal) < y \equiv x \leq y \wedge x \neq y$

**instance**
⟨*proof*⟩

**end**

Variants of min-plus and max-plus semirings can easily be obtained. Here we formalise the min-plus semiring over the natural numbers as an example.

**datatype** *pnat* = *pnat nat* | *PInfty* — plus infinity

**fun** *pnat-min* **where**
*pnat-min* (*pnat x*) (*pnat y*) = *pnat* (*min x y*)
| *pnat-min x PInfty* = *x*
| *pnat-min PInfty x* = *x*

**lemma** *pnat-min-simp-3* [*simp*]: *pnat-min PInfty y* = *y*
⟨*proof*⟩

**fun** *pnat-plus* **where**
*pnat-plus* (*pnat x*) (*pnat y*) = *pnat* (*x* + *y*)
| *pnat-plus* - - = *PInfty*

**instantiation** *pnat* :: *selective-semiring*
**begin**

**definition** *zero-pnat-def*:
*0* ≡ *PInfty*

**definition** *one-pnat-def*:
*1* ≡ *pnat 0*

**definition** *plus-pnat-def*:
*x* + *y* ≡ *pnat-min x y*

**definition** *times-pnat-def*:
*x* ∗ *y* ≡ *pnat-plus x y*

**definition** *less-eq-pnat-def*:
$(x::pnat) \leq y \equiv x + y = y$

**definition** *less-pnat-def*:
$(x::pnat) < y \equiv x \leq y \wedge x \neq y$

**lemma** *zero-pnat-top*: $(x::pnat) \leq 1$
⟨*proof*⟩

**instance**
⟨*proof*⟩

**end**

**end**

# 5 Matrices

**theory** *Matrix*
**imports** *HOL−Word.Word Dioid*
**begin**

In this section we formalise a perhaps more natural version of matrices of fixed dimension ($m \times n$-matrices). It is well known that such matrices over a Kleene algebra form a Kleene algebra [8].

## 5.1 Type Definition

**typedef** (**overloaded**) *'a atMost* = {..<*len-of TYPE*('a::*len*)}
⟨*proof*⟩

**declare** *Rep-atMost-inject* [*simp*]

**lemma** *UNIV-atMost*:
(*UNIV*::*'a atMost set*) = *Abs-atMost* ' {..<*len-of TYPE*('a::*len*)}
⟨*proof*⟩

**lemma** *finite-UNIV-atMost* [*simp*]: *finite* (*UNIV*::('a::*len*) *atMost set*)
⟨*proof*⟩

Our matrix type is similar to *'a^'n^'m* from *HOL/Multivariate_Analysis/Finite_Cartesian_Product.thy* but (i) we explicitly define a type constructor for matrices and square matrices, and (ii) in the definition of operations, e.g., matrix multiplication, we impose weaker sort requirements on the element type.

**context notes** [[*typedef-overloaded*]]
**begin**

**datatype** ('a,'m,'n) *matrix* = *Matrix 'm atMost* ⇒ *'n atMost* ⇒ *'a*

**datatype** ('a,'m) *sqmatrix* = *SqMatrix 'm atMost* ⇒ *'m atMost* ⇒ *'a*

**end**

**fun** *sqmatrix-of-matrix* **where**

*sqmatrix-of-matrix* (*Matrix A*) = *SqMatrix A*

**fun** *matrix-of-sqmatrix* **where**
*matrix-of-sqmatrix* (*SqMatrix A*) = *Matrix A*

## 5.2 0 and 1

**instantiation** *matrix* :: (*zero*,*type*,*type*) *zero*
**begin**
**definition** *zero-matrix-def*: *0* ≡ *Matrix* (λ*i j*. *0*)
**instance** ⟨*proof*⟩
**end**

**instantiation** *sqmatrix* :: (*zero*,*type*) *zero*
**begin**
**definition** *zero-sqmatrix-def*: *0* ≡ *SqMatrix* (λ*i j*. *0*)
**instance** ⟨*proof*⟩
**end**

Tricky sort issues: compare *one-matrix* with *one-sqmatrix* ...

**instantiation** *matrix* :: ({*zero*,*one*},*len*,*len*) *one*
**begin**
**definition** *one-matrix-def*:
*1* ≡ *Matrix* (λ*i j*. *if Rep-atMost i* = *Rep-atMost j then 1 else 0*)
**instance** ⟨*proof*⟩
**end**

**instantiation** *sqmatrix* :: ({*zero*,*one*},*type*) *one*
**begin**
**definition** *one-sqmatrix-def*:
*1* ≡ *SqMatrix* (λ*i j*. *if i* = *j then 1 else 0*)
**instance** ⟨*proof*⟩
**end**

## 5.3 Matrix Addition

**fun** *matrix-plus* **where**
*matrix-plus* (*Matrix A*) (*Matrix B*) = *Matrix* (λ*i j*. *A i j* + *B i j*)

**instantiation** *matrix* :: (*plus*,*type*,*type*) *plus*
**begin**
**definition** *plus-matrix-def*: *A* + *B* ≡ *matrix-plus A B*
**instance** ⟨*proof*⟩
**end**

**lemma** *plus-matrix-def'* [*simp*]:
*Matrix A* + *Matrix B* = *Matrix* (λ*i j*. *A i j* + *B i j*)
⟨*proof*⟩

**instantiation** *sqmatrix* :: (*plus*,*type*) *plus*
**begin**
**definition** *plus-sqmatrix-def*:
$A + B \equiv$ *sqmatrix-of-matrix* (*matrix-of-sqmatrix* $A$ + *matrix-of-sqmatrix* $B$)
**instance** ⟨*proof*⟩
**end**

**lemma** *plus-sqmatrix-def'* [*simp*]:
*SqMatrix* $A$ + *SqMatrix* $B$ = *SqMatrix* ($\lambda i\ j.\ A\ i\ j + B\ i\ j$)
⟨*proof*⟩

**lemma** *matrix-add-0-right* [*simp*]:
$A + 0 = (A::('a::monoid\text{-}add,'m,'n)\ matrix)$
⟨*proof*⟩

**lemma** *matrix-add-0-left* [*simp*]:
$0 + A = (A::('a::monoid\text{-}add,'m,'n)\ matrix)$
⟨*proof*⟩

**lemma** *matrix-add-commute* [*simp*]:
$(A::('a::ab\text{-}semigroup\text{-}add,'m,'n)\ matrix) + B = B + A$
⟨*proof*⟩

**lemma** *matrix-add-assoc*:
$(A::('a::semigroup\text{-}add,'m,'n)\ matrix) + B + C = A + (B + C)$
⟨*proof*⟩

**lemma** *matrix-add-left-commute* [*simp*]:
$(A::('a::ab\text{-}semigroup\text{-}add,'m,'n)\ matrix) + (B + C) = B + (A + C)$
⟨*proof*⟩

**lemma** *sqmatrix-add-0-right* [*simp*]:
$A + 0 = (A::('a::monoid\text{-}add,'m)\ sqmatrix)$
⟨*proof*⟩

**lemma** *sqmatrix-add-0-left* [*simp*]:
$0 + A = (A::('a::monoid\text{-}add,'m)\ sqmatrix)$
⟨*proof*⟩

**lemma** *sqmatrix-add-commute* [*simp*]:
$(A::('a::ab\text{-}semigroup\text{-}add,'m)\ sqmatrix) + B = B + A$
⟨*proof*⟩

**lemma** *sqmatrix-add-assoc*:
$(A::('a::semigroup\text{-}add,'m)\ sqmatrix) + B + C = A + (B + C)$
⟨*proof*⟩

**lemma** *sqmatrix-add-left-commute* [*simp*]:
$(A::('a::ab\text{-}semigroup\text{-}add,'m)\ sqmatrix) + (B + C) = B + (A + C)$

⟨*proof*⟩

## 5.4 Order (via Addition)

**instantiation** *matrix* :: (*plus*,*type*,*type*) *plus-ord*
**begin**
**definition** *less-eq-matrix-def*:
$(A::('a,\ 'b,\ 'c)\ matrix) \leq B \equiv A + B = B$
**definition** *less-matrix-def*:
$(A::('a,\ 'b,\ 'c)\ matrix) < B \equiv A \leq B \wedge A \neq B$

**instance**
⟨*proof*⟩
**end**

**instantiation** *sqmatrix* :: (*plus*,*type*) *plus-ord*
**begin**
**definition** *less-eq-sqmatrix-def*:
$(A::('a,\ 'b)\ sqmatrix) \leq B \equiv A + B = B$
**definition** *less-sqmatrix-def*:
$(A::('a,\ 'b)\ sqmatrix) < B \equiv A \leq B \wedge A \neq B$

**instance**
⟨*proof*⟩
**end**

## 5.5 Matrix Multiplication

**fun** *matrix-times* :: ('*a*::{*comm-monoid-add*,*times*},'*m*,'*k*) *matrix* ⇒ ('*a*,'*k*,'*n*) *matrix* ⇒ ('*a*,'*m*,'*n*) *matrix* **where**
*matrix-times* (*Matrix A*) (*Matrix B*) = *Matrix* ($\lambda i\ j.$ *sum* ($\lambda k.$ *A i k* ∗ *B k j*) (*UNIV*::'*k atMost set*))

**notation** *matrix-times* (**infixl** $*_M$ *70*)

**instantiation** *sqmatrix* :: ({*comm-monoid-add*,*times*},*type*) *times*
**begin**
**definition** *times-sqmatrix-def*:
*A* ∗ *B* = *sqmatrix-of-matrix* (*matrix-of-sqmatrix A* $*_M$ *matrix-of-sqmatrix B*)
**instance** ⟨*proof*⟩
**end**

**lemma** *times-sqmatrix-def'* [*simp*]:
*SqMatrix A* ∗ *SqMatrix B* = *SqMatrix* ($\lambda i\ j.$ *sum* ($\lambda k.$ *A i k* ∗ *B k j*) (*UNIV*::'*k atMost set*))
⟨*proof*⟩

**lemma** *matrix-mult-0-right* [*simp*]:
(*A*::('*a*::{*comm-monoid-add*,*mult-zero*},'*m*,'*n*) *matrix*) $*_M$ *0* = *0*
⟨*proof*⟩

**lemma** *matrix-mult-0-left* [*simp*]:
$0 *_M (A::('a::\{comm\text{-}monoid\text{-}add,mult\text{-}zero\},'m,'n)\ matrix) = 0$
⟨*proof*⟩

**lemma** *sum-delta-r-0* [*simp*]:
$[\![ finite\ S; j \notin S ]\!] \Longrightarrow (\sum k{\in}S.\ f\,k * (if\ k = j\ then\ 1\ else\ (0::'b::\{semiring\text{-}0,monoid\text{-}mult\}))) = 0$
⟨*proof*⟩

**lemma** *sum-delta-r-1* [*simp*]:
$[\![ finite\ S; j \in S ]\!] \Longrightarrow (\sum k{\in}S.\ f\,k * (if\ k = j\ then\ 1\ else\ (0::'b::\{semiring\text{-}0,monoid\text{-}mult\}))) = f\ j$
⟨*proof*⟩

**lemma** *matrix-mult-1-right* [*simp*]:
$(A::('a::\{semiring\text{-}0,monoid\text{-}mult\},'m::len,'n::len)\ matrix) *_M\ 1 = A$
⟨*proof*⟩

**lemma** *sum-delta-l-0* [*simp*]:
$[\![ finite\ S; i \notin S ]\!] \Longrightarrow (\sum k{\in}S.\ (if\ i = k\ then\ 1\ else\ (0::'b::\{semiring\text{-}0,monoid\text{-}mult\})) * f\ k\ j) = 0$
⟨*proof*⟩

**lemma** *sum-delta-l-1* [*simp*]:
$[\![ finite\ S; i \in S ]\!] \Longrightarrow (\sum k{\in}S.\ (if\ i = k\ then\ 1\ else\ (0::'b::\{semiring\text{-}0,monoid\text{-}mult\})) * f\ k\ j) = f\ i\ j$
⟨*proof*⟩

**lemma** *matrix-mult-1-left* [*simp*]:
$1 *_M (A::('a::\{semiring\text{-}0,monoid\text{-}mult\},'m::len,'n::len)\ matrix) = A$
⟨*proof*⟩

**lemma** *matrix-mult-assoc*:
$(A::('a::semiring\text{-}0,'m,'n)\ matrix) *_M\ B *_M\ C = A *_M (B *_M\ C)$
⟨*proof*⟩

**lemma** *matrix-mult-distrib-left*:
$(A::('a::\{comm\text{-}monoid\text{-}add,semiring\},'m,'n::len)\ matrix) *_M (B + C) = A *_M B + A *_M\ C$
⟨*proof*⟩

**lemma** *matrix-mult-distrib-right*:
$((A::('a::\{comm\text{-}monoid\text{-}add,semiring\},'m,'n::len)\ matrix) + B) *_M\ C = A *_M C + B *_M\ C$
⟨*proof*⟩

**lemma** *sqmatrix-mult-0-right* [*simp*]:
$(A::('a::\{comm\text{-}monoid\text{-}add,mult\text{-}zero\},'m)\ sqmatrix) * 0 = 0$

⟨*proof*⟩

**lemma** *sqmatrix-mult-0-left* [*simp*]:
*0* ∗ (*A*::('*a*::{*comm-monoid-add*,*mult-zero*},'*m*) *sqmatrix*) = *0*
⟨*proof*⟩

**lemma** *sqmatrix-mult-1-right* [*simp*]:
(*A*::('*a*::{*semiring-0*,*monoid-mult*},'*m*::*len*) *sqmatrix*) ∗ *1* = *A*
⟨*proof*⟩

**lemma** *sqmatrix-mult-1-left* [*simp*]:
*1* ∗ (*A*::('*a*::{*semiring-0*,*monoid-mult*},'*m*::*len*) *sqmatrix*) = *A*
⟨*proof*⟩

**lemma** *sqmatrix-mult-assoc*:
(*A*::('*a*::{*semiring-0*,*monoid-mult*},'*m*) *sqmatrix*) ∗ *B* ∗ *C* = *A* ∗ (*B* ∗ *C*)
⟨*proof*⟩

**lemma** *sqmatrix-mult-distrib-left*:
(*A*::('*a*::{*comm-monoid-add*,*semiring*},'*m*::*len*) *sqmatrix*) ∗ (*B* + *C*) = *A* ∗ *B* + *A* ∗ *C*
⟨*proof*⟩

**lemma** *sqmatrix-mult-distrib-right*:
((*A*::('*a*::{*comm-monoid-add*,*semiring*},'*m*::*len*) *sqmatrix*) + *B*) ∗ *C* = *A* ∗ *C* + *B* ∗ *C*
⟨*proof*⟩

## 5.6 Square-Matrix Model of Dioids

The following subclass proofs are necessary to connect parts of our algebraic hierarchy to the hierarchy found in the Isabelle/HOL library.

**subclass** (**in** *ab-near-semiring-one-zerol*) *comm-monoid-add*
⟨*proof*⟩

**subclass** (**in** *semiring-one-zero*) *semiring-0*
⟨*proof*⟩

**subclass** (**in** *ab-near-semiring-one*) *monoid-mult* ⟨*proof*⟩

**instantiation** *sqmatrix* :: (*dioid-one-zero*,*len*) *dioid-one-zero*
**begin**
**instance**
⟨*proof*⟩
**end**

### 5.7 Kleene Star for Matrices

We currently do not implement the Kleene star of matrices, since this is complicated.

**end**

## 6 Conway Algebras

**theory** *Conway*
**imports** *Dioid*
**begin**

We define a weak regular algebra which can serve as a common basis for Kleene algebra and demonic reginement algebra. It is closely related to an axiomatisation given by Conway [8].

**class** *dagger-op* =
**fixes** *dagger* :: *'a* ⇒ *'a* (-† [*101*] *100*)

### 6.1 Near Conway Algebras

**class** *near-conway-base* = *near-dioid-one* + *dagger-op* +
**assumes** *dagger-denest*: $(x + y)^\dagger = (x^\dagger \cdot y)^\dagger \cdot x^\dagger$
**and** *dagger-prod-unfold* [*simp*]: $1 + x \cdot (y \cdot x)^\dagger \cdot y = (x \cdot y)^\dagger$

**begin**

**lemma** *dagger-unfoldl-eq* [*simp*]: $1 + x \cdot x^\dagger = x^\dagger$
⟨*proof*⟩

**lemma** *dagger-unfoldl*: $1 + x \cdot x^\dagger \leq x^\dagger$
⟨*proof*⟩

**lemma** *dagger-unfoldr-eq* [*simp*]: $1 + x^\dagger \cdot x = x^\dagger$
⟨*proof*⟩

**lemma** *dagger-unfoldr*: $1 + x^\dagger \cdot x \leq x^\dagger$
⟨*proof*⟩

**lemma** *dagger-unfoldl-distr* [*simp*]: $y + x \cdot x^\dagger \cdot y = x^\dagger \cdot y$
⟨*proof*⟩

**lemma** *dagger-unfoldr-distr* [*simp*]: $y + x^\dagger \cdot x \cdot y = x^\dagger \cdot y$
⟨*proof*⟩

**lemma** *dagger-refl*: $1 \leq x^\dagger$
⟨*proof*⟩

**lemma** *dagger-plus-one* [*simp*]: $1 + x^\dagger = x^\dagger$

⟨*proof*⟩

**lemma** *star-1l*: $x \cdot x^\dagger \leq x^\dagger$
⟨*proof*⟩

**lemma** *star-1r*: $x^\dagger \cdot x \leq x^\dagger$
⟨*proof*⟩

**lemma** *dagger-ext*: $x \leq x^\dagger$
⟨*proof*⟩

**lemma** *dagger-trans-eq* [*simp*]: $x^\dagger \cdot x^\dagger = x^\dagger$
⟨*proof*⟩

**lemma** *dagger-subdist*: $x^\dagger \leq (x + y)^\dagger$
⟨*proof*⟩

**lemma** *dagger-subdist-var*: $x^\dagger + y^\dagger \leq (x + y)^\dagger$
⟨*proof*⟩

**lemma** *dagger-iso* [*intro*]: $x \leq y \Longrightarrow x^\dagger \leq y^\dagger$
⟨*proof*⟩

**lemma** *star-square*: $(x \cdot x)^\dagger \leq x^\dagger$
⟨*proof*⟩

**lemma** *dagger-rtc1-eq* [*simp*]: $1 + x + x^\dagger \cdot x^\dagger = x^\dagger$
⟨*proof*⟩

Nitpick refutes the next lemmas.

**lemma** $y + y \cdot x^\dagger \cdot x = y \cdot x^\dagger$

⟨*proof*⟩

**lemma** $y \cdot x^\dagger = y + y \cdot x \cdot x^\dagger$

⟨*proof*⟩

**lemma** $(x + y)^\dagger = x^\dagger \cdot (y \cdot x^\dagger)^\dagger$

⟨*proof*⟩

**lemma** $(x^\dagger)^\dagger = x^\dagger$

⟨*proof*⟩

**lemma** $(1 + x)^\star = x^\star$

⟨*proof*⟩

**lemma** $x^\dagger \cdot x = x \cdot x^\dagger$

⟨*proof*⟩

**end**

## 6.2 Pre-Conway Algebras

**class** *pre-conway-base* = *near-conway-base* + *pre-dioid-one*

**begin**

**lemma** *dagger-subdist-var-3*: $x^\dagger \cdot y^\dagger \leq (x + y)^\dagger$
⟨*proof*⟩

**lemma** *dagger-subdist-var-2*: $x \cdot y \leq (x + y)^\dagger$
⟨*proof*⟩

**lemma** *dagger-sum-unfold* [*simp*]: $x^\dagger + x^\dagger \cdot y \cdot (x + y)^\dagger = (x + y)^\dagger$
⟨*proof*⟩

**end**

## 6.3 Conway Algebras

**class** *conway-base* = *pre-conway-base* + *dioid-one*

**begin**

**lemma** *troeger*: $(x + y)^\dagger \cdot z = x^\dagger \cdot (y \cdot (x + y)^\dagger \cdot z + z)$
⟨*proof*⟩

**lemma** *dagger-slide-var1*: $x^\dagger \cdot x \leq x \cdot x^\dagger$
⟨*proof*⟩

**lemma** *dagger-slide-var1-eq*: $x^\dagger \cdot x = x \cdot x^\dagger$
⟨*proof*⟩

**lemma** *dagger-slide-eq*: $(x \cdot y)^\dagger \cdot x = x \cdot (y \cdot x)^\dagger$
⟨*proof*⟩

**end**

## 6.4 Conway Algebras with Zero

**class** *near-conway-base-zerol* = *near-conway-base* + *near-dioid-one-zerol*

**begin**

**lemma** *dagger-annil* [*simp*]: $1 + x \cdot 0 = (x \cdot 0)^\dagger$
⟨*proof*⟩

**lemma** *zero-dagger* [*simp*]: $0^\dagger = 1$
⟨*proof*⟩

**end**

**class** *pre-conway-base-zerol* = *near-conway-base-zerol* + *pre-dioid*

**class** *conway-base-zerol* = *pre-conway-base-zerol* + *dioid*

**subclass** (**in** *pre-conway-base-zerol*) *pre-conway-base* ⟨*proof*⟩

**subclass** (**in** *conway-base-zerol*) *conway-base* ⟨*proof*⟩

**context** *conway-base-zerol*
**begin**

**lemma** $z \cdot x \leq y \cdot z \Longrightarrow z \cdot x^\dagger \leq y^\dagger \cdot z$

⟨*proof*⟩

**end**

## 6.5 Conway Algebras with Simulation

**class** *near-conway* = *near-conway-base* +
**assumes** *dagger-simr*: $z \cdot x \leq y \cdot z \Longrightarrow z \cdot x^\dagger \leq y^\dagger \cdot z$

**begin**

**lemma** *dagger-slide-var*: $x \cdot (y \cdot x)^\dagger \leq (x \cdot y)^\dagger \cdot x$
⟨*proof*⟩

Nitpick refutes the next lemma.

**lemma** *dagger-slide*: $x \cdot (y \cdot x)^\dagger = (x \cdot y)^\dagger \cdot x$

⟨*proof*⟩

We say that $y$ preserves $x$ if $x \cdot y \cdot x = x \cdot y$ and $!x \cdot y \cdot !x = !x \cdot y$. This definition is taken from Solin [26]. It is useful for program transformation.

**lemma** *preservation1*: $x \cdot y \leq x \cdot y \cdot x \Longrightarrow x \cdot y^\dagger \leq (x \cdot y + z)^\dagger \cdot x$
⟨*proof*⟩

**end**

**class** *near-conway-zerol* = *near-conway* + *near-dioid-one-zerol*

**class** *pre-conway* = *near-conway* + *pre-dioid-one*

**begin**

**subclass** *pre-conway-base* ⟨*proof*⟩

**lemma** *dagger-slide*: $x \cdot (y \cdot x)^\dagger = (x \cdot y)^\dagger \cdot x$
⟨*proof*⟩

**lemma** *dagger-denest2*: $(x + y)^\dagger = x^\dagger \cdot (y \cdot x^\dagger)^\dagger$
⟨*proof*⟩

**lemma** *preservation2*: $y \cdot x \leq y \Longrightarrow (x \cdot y)^\dagger \cdot x \leq x \cdot y^\dagger$
⟨*proof*⟩

**lemma** *preservation1-eq*: $x \cdot y \leq x \cdot y \cdot x \Longrightarrow y \cdot x \leq y \Longrightarrow (x \cdot y)^\dagger \cdot x = x \cdot y^\dagger$
⟨*proof*⟩

**end**

**class** *pre-conway-zerol* = *near-conway-zerol* + *pre-dioid-one-zerol*

**begin**

**subclass** *pre-conway* ⟨*proof*⟩

**end**

**class** *conway* = *pre-conway* + *dioid-one*

**class** *conway-zerol* = *pre-conway* + *dioid-one-zerol*

**begin**

**subclass** *conway-base* ⟨*proof*⟩

Nitpick refutes the next lemmas.

**lemma** $1 = 1^\dagger$

⟨*proof*⟩

**lemma** $(x^\dagger)^\dagger = x^\dagger$

⟨*proof*⟩

**lemma** *dagger-denest-var* [*simp*]: $(x + y)^\dagger = (x^\dagger \cdot y^\dagger)^\dagger$

⟨*proof*⟩

**lemma** *star2* [*simp*]: $(1 + x)^\dagger = x^\dagger$

⟨*proof*⟩

**end**

**end**

# 7 Kleene Algebras

**theory** *Kleene-Algebra*
**imports** *Conway*
**begin**

## 7.1 Left Near Kleene Algebras

Extending the hierarchy developed in *Dioid* we now add an operation of Kleene star, finite iteration, or reflexive transitive closure to variants of Dioids. Since a multiplicative unit is needed for defining the star we only consider variants with 1; 0 can be added separately. We consider the left star induction axiom and the right star induction axiom independently since in some applications, e.g., Salomaa's axioms, probabilistic Kleene algebras, or completeness proofs with respect to the equational theoy of regular expressions and regular languages, the right star induction axiom is not needed or not valid.

We start with near dioids, then consider pre-dioids and finally dioids. It turns out that many of the known laws of Kleene algebras hold already in these more general settings. In fact, all our equational theorems have been proved within left Kleene algebras, as expected.

Although most of the proofs in this file could be fully automated by Sledgehammer and Metis, we display step-wise proofs as they would appear in a text book. First, this file may then be useful as a reference manual on Kleene algebra. Second, it is better protected against changes in the underlying theories and supports easy translation of proofs into other settings.

**class** *left-near-kleene-algebra* = *near-dioid-one* + *star-op* +
  **assumes** *star-unfoldl*: $1 + x \cdot x^\star \leq x^\star$
  **and** *star-inductl*: $z + x \cdot y \leq y \Longrightarrow x^\star \cdot z \leq y$

**begin**

First we prove two immediate consequences of the unfold axiom. The first one states that starred elements are reflexive.

**lemma** *star-ref* [*simp*]: $1 \leq x^\star$
  ⟨*proof*⟩

Reflexivity of starred elements implies, by definition of the order, that 1 is an additive unit for starred elements.

**lemma** *star-plus-one* [*simp*]: $1 + x^\star = x^\star$
⟨*proof*⟩

**lemma** *star-1l* [*simp*]: $x \cdot x^\star \leq x^\star$
⟨*proof*⟩

**lemma** $x^\star \cdot x \leq x^\star$

⟨*proof*⟩

**lemma** $x \cdot x^\star = x^\star$

⟨*proof*⟩

Next we show that starred elements are transitive.

**lemma** *star-trans-eq* [*simp*]: $x^\star \cdot x^\star = x^\star$
⟨*proof*⟩

**lemma** *star-trans*: $x^\star \cdot x^\star \leq x^\star$
⟨*proof*⟩

We now derive variants of the star induction axiom.

**lemma** *star-inductl-var*: $x \cdot y \leq y \Longrightarrow x^\star \cdot y \leq y$
⟨*proof*⟩

**lemma** *star-inductl-var-equiv* [*simp*]: $x^\star \cdot y \leq y \longleftrightarrow x \cdot y \leq y$
⟨*proof*⟩

**lemma** *star-inductl-var-eq*: $x \cdot y = y \Longrightarrow x^\star \cdot y \leq y$
⟨*proof*⟩

**lemma** *star-inductl-var-eq2*: $y = x \cdot y \Longrightarrow y = x^\star \cdot y$
⟨*proof*⟩

**lemma** $y = x \cdot y \longleftrightarrow y = x^\star \cdot y$

⟨*proof*⟩

**lemma** $x^\star \cdot z \leq y \Longrightarrow z + x \cdot y \leq y$

⟨*proof*⟩

**lemma** *star-inductl-one*: $1 + x \cdot y \leq y \Longrightarrow x^\star \leq y$
⟨*proof*⟩

**lemma** *star-inductl-star*: $x \cdot y^\star \leq y^\star \Longrightarrow x^\star \leq y^\star$

⟨*proof*⟩

**lemma** *star-inductl-eq*: $z + x \cdot y = y \Longrightarrow x^\star \cdot z \leq y$
⟨*proof*⟩

We now prove two facts related to 1.

**lemma** *star-subid*: $x \leq 1 \Longrightarrow x^\star = 1$
⟨*proof*⟩

**lemma** *star-one* [*simp*]: $1^\star = 1$
⟨*proof*⟩

We now prove a subdistributivity property for the star (which is equivalent to isotonicity of star).

**lemma** *star-subdist*: $x^\star \leq (x + y)^\star$
⟨*proof*⟩

**lemma** *star-subdist-var*: $x^\star + y^\star \leq (x + y)^\star$
⟨*proof*⟩

**lemma** *star-iso* [*intro*]: $x \leq y \Longrightarrow x^\star \leq y^\star$
⟨*proof*⟩

We now prove some more simple properties.

**lemma** *star-invol* [*simp*]: $(x^\star)^\star = x^\star$
⟨*proof*⟩

**lemma** *star2* [*simp*]: $(1 + x)^\star = x^\star$
⟨*proof*⟩

**lemma** $1 + x^\star \cdot x \leq x^\star$

⟨*proof*⟩

**lemma** $x \leq x^\star$

⟨*proof*⟩

**lemma** $x^\star \cdot x \leq x^\star$

⟨*proof*⟩

**lemma** $1 + x \cdot x^\star = x^\star$

⟨*proof*⟩

**lemma** $x \cdot z \leq z \cdot y \Longrightarrow x^\star \cdot z \leq z \cdot y^\star$

⟨*proof*⟩

The following facts express inductive conditions that are used to show that $(x + y)^\star$ is the greatest term that can be built from $x$ and $y$.

**lemma** *prod-star-closure*: $x \leq z^\star \Longrightarrow y \leq z^\star \Longrightarrow x \cdot y \leq z^\star$
⟨*proof*⟩

**lemma** *star-star-closure*: $x^\star \leq z^\star \Longrightarrow (x^\star)^\star \leq z^\star$
⟨*proof*⟩

**lemma** *star-closed-unfold*: $x^\star = x \Longrightarrow x = 1 + x \cdot x$
⟨*proof*⟩

**lemma** $x^\star = x \longleftrightarrow x = 1 + x \cdot x$

⟨*proof*⟩

**end**

## 7.2 Left Pre-Kleene Algebras

**class** *left-pre-kleene-algebra* = *left-near-kleene-algebra* + *pre-dioid-one*

**begin**

We first prove that the star operation is extensive.

**lemma** *star-ext* [*simp*]: $x \leq x^\star$
⟨*proof*⟩

We now prove a right star unfold law.

**lemma** *star-1r* [*simp*]: $x^\star \cdot x \leq x^\star$
⟨*proof*⟩

**lemma** *star-unfoldr*: $1 + x^\star \cdot x \leq x^\star$
⟨*proof*⟩

**lemma** $1 + x^\star \cdot x = x^\star$

⟨*proof*⟩

Next we prove a simulation law for the star. It is instrumental in proving further properties.

**lemma** *star-sim1*: $x \cdot z \leq z \cdot y \Longrightarrow x^\star \cdot z \leq z \cdot y^\star$
⟨*proof*⟩

The next lemma is used in omega algebras to prove, for instance, Bachmair and Dershowitz's separation of termination theorem [4]. The property at the left-hand side of the equivalence is known as *quasicommutation*.

**lemma** *quasicomm-var*: $y \cdot x \leq x \cdot (x + y)^\star \longleftrightarrow y^\star \cdot x \leq x \cdot (x + y)^\star$

⟨*proof*⟩

**lemma** *star-slide1*: $(x \cdot y)^\star \cdot x \leq x \cdot (y \cdot x)^\star$
⟨*proof*⟩

**lemma** $(x \cdot y)^\star \cdot x = x \cdot (y \cdot x)^\star$

⟨*proof*⟩

**lemma** *star-slide-var1*: $x^\star \cdot x \leq x \cdot x^\star$
⟨*proof*⟩

We now show that the (left) star unfold axiom can be strengthened to an equality.

**lemma** *star-unfoldl-eq* [*simp*]: $1 + x \cdot x^\star = x^\star$
⟨*proof*⟩

**lemma** $1 + x^\star \cdot x = x^\star$

⟨*proof*⟩

Next we relate the star and the reflexive transitive closure operation.

**lemma** *star-rtc1-eq* [*simp*]: $1 + x + x^\star \cdot x^\star = x^\star$
⟨*proof*⟩

**lemma** *star-rtc1*: $1 + x + x^\star \cdot x^\star \leq x^\star$
⟨*proof*⟩

**lemma** *star-rtc2*: $1 + x \cdot x \leq x \longleftrightarrow x = x^\star$
⟨*proof*⟩

**lemma** *star-rtc3*: $1 + x \cdot x = x \longleftrightarrow x = x^\star$
⟨*proof*⟩

**lemma** *star-rtc-least*: $1 + x + y \cdot y \leq y \Longrightarrow x^\star \leq y$
⟨*proof*⟩

**lemma** *star-rtc-least-eq*: $1 + x + y \cdot y = y \Longrightarrow x^\star \leq y$
⟨*proof*⟩

**lemma** $1 + x + y \cdot y \leq y \longleftrightarrow x^\star \leq y$

⟨*proof*⟩

The next lemmas are again related to closure conditions

**lemma** *star-subdist-var-1*: $x \leq (x + y)^\star$
⟨*proof*⟩

**lemma** *star-subdist-var-2*: $x \cdot y \leq (x + y)^\star$

⟨*proof*⟩

**lemma** *star-subdist-var-3*: $x^\star \cdot y^\star \leq (x + y)^\star$
⟨*proof*⟩

We now prove variants of sum-elimination laws under a star. These are also known a denesting laws or as sum-star laws.

**lemma** *star-denest* [*simp*]: $(x + y)^\star = (x^\star \cdot y^\star)^\star$
⟨*proof*⟩

**lemma** *star-sum-var* [*simp*]: $(x^\star + y^\star)^\star = (x + y)^\star$
⟨*proof*⟩

**lemma** *star-denest-var* [*simp*]: $x^\star \cdot (y \cdot x^\star)^\star = (x + y)^\star$
⟨*proof*⟩

**lemma** *star-denest-var-2* [*simp*]: $x^\star \cdot (y \cdot x^\star)^\star = (x^\star \cdot y^\star)^\star$
⟨*proof*⟩

**lemma** *star-denest-var-3* [*simp*]: $x^\star \cdot (y^\star \cdot x^\star)^\star = (x^\star \cdot y^\star)^\star$
⟨*proof*⟩

**lemma** *star-denest-var-4* [*ac-simps*]: $(y^\star \cdot x^\star)^\star = (x^\star \cdot y^\star)^\star$
⟨*proof*⟩

**lemma** *star-denest-var-5* [*ac-simps*]: $x^\star \cdot (y \cdot x^\star)^\star = y^\star \cdot (x \cdot y^\star)^\star$
⟨*proof*⟩

**lemma** $x^\star \cdot (y \cdot x^\star)^\star = (x^\star \cdot y)^\star \cdot x^\star$

⟨*proof*⟩

**lemma** *star-denest-var-6* [*simp*]: $x^\star \cdot y^\star \cdot (x + y)^\star = (x + y)^\star$
⟨*proof*⟩

**lemma** *star-denest-var-7* [*simp*]: $(x + y)^\star \cdot x^\star \cdot y^\star = (x + y)^\star$
⟨*proof*⟩

**lemma** *star-denest-var-8* [*simp*]: $x^\star \cdot y^\star \cdot (x^\star \cdot y^\star)^\star = (x^\star \cdot y^\star)^\star$
⟨*proof*⟩

**lemma** *star-denest-var-9* [*simp*]: $(x^\star \cdot y^\star)^\star \cdot x^\star \cdot y^\star = (x^\star \cdot y^\star)^\star$
⟨*proof*⟩

The following statements are well known from term rewriting. They are all variants of the Church-Rosser theorem in Kleene algebra [27]. But first we prove a law relating two confluence properties.

**lemma** *confluence-var* [*iff*]: $y \cdot x^\star \leq x^\star \cdot y^\star \longleftrightarrow y^\star \cdot x^\star \leq x^\star \cdot y^\star$

⟨*proof*⟩

**lemma** *church-rosser* [*intro*]: $y^\star \cdot x^\star \leq x^\star \cdot y^\star \Longrightarrow (x + y)^\star = x^\star \cdot y^\star$
⟨*proof*⟩

**lemma** *church-rosser-var*: $y \cdot x^\star \leq x^\star \cdot y^\star \Longrightarrow (x + y)^\star = x^\star \cdot y^\star$
⟨*proof*⟩

**lemma** *church-rosser-to-confluence*: $(x + y)^\star \leq x^\star \cdot y^\star \Longrightarrow y^\star \cdot x^\star \leq x^\star \cdot y^\star$
⟨*proof*⟩

**lemma** *church-rosser-equiv*: $y^\star \cdot x^\star \leq x^\star \cdot y^\star \longleftrightarrow (x + y)^\star = x^\star \cdot y^\star$
⟨*proof*⟩

**lemma** *confluence-to-local-confluence*: $y^\star \cdot x^\star \leq x^\star \cdot y^\star \Longrightarrow y \cdot x \leq x^\star \cdot y^\star$
⟨*proof*⟩

**lemma** $y \cdot x \leq x^\star \cdot y^\star \Longrightarrow y^\star \cdot x^\star \leq x^\star \cdot y^\star$

⟨*proof*⟩

**lemma** $y \cdot x \leq x^\star \cdot y^\star \Longrightarrow (x + y)^\star \leq x^\star \cdot y^\star$
⟨*proof*⟩

More variations could easily be proved. The last counterexample shows that Newman's lemma needs a wellfoundedness assumption. This is well known.

The next lemmas relate the reflexive transitive closure and the transitive closure.

**lemma** *sup-id-star1*: $1 \leq x \Longrightarrow x \cdot x^\star = x^\star$
⟨*proof*⟩

**lemma** *sup-id-star2*: $1 \leq x \Longrightarrow x^\star \cdot x = x^\star$
⟨*proof*⟩

**lemma** $1 + x^\star \cdot x = x^\star$

⟨*proof*⟩

**lemma** $(x \cdot y)^\star \cdot x = x \cdot (y \cdot x)^\star$

⟨*proof*⟩

**lemma** $x \cdot x = x \Longrightarrow x^\star = 1 + x$

⟨*proof*⟩

**end**

## 7.3 Left Kleene Algebras

**class** *left-kleene-algebra* = *left-pre-kleene-algebra* + *dioid-one*

**begin**

In left Kleene algebras the non-fact $z + y \cdot x \leq y \Longrightarrow z \cdot x^\star \leq y$ is a good challenge for counterexample generators. A model of left Kleene algebras in which the right star induction law does not hold has been given by Kozen [20].

We now show that the right unfold law becomes an equality.

**lemma** *star-unfoldr-eq* [*simp*]: $1 + x^\star \cdot x = x^\star$
⟨*proof*⟩

The following more complex unfold law has been used as an axiom, called prodstar, by Conway [8].

**lemma** *star-prod-unfold* [*simp*]: $1 + x \cdot (y \cdot x)^\star \cdot y = (x \cdot y)^\star$
⟨*proof*⟩

The slide laws, which have previously been inequalities, now become equations.

**lemma** *star-slide* [*ac-simps*]: $(x \cdot y)^\star \cdot x = x \cdot (y \cdot x)^\star$
⟨*proof*⟩

**lemma** *star-slide-var* [*ac-simps*]: $x^\star \cdot x = x \cdot x^\star$
⟨*proof*⟩

**lemma** *star-sum-unfold-var* [*simp*]: $1 + x^\star \cdot (x + y)^\star \cdot y^\star = (x + y)^\star$
⟨*proof*⟩

The following law shows how starred sums can be unfolded.

**lemma** *star-sum-unfold* [*simp*]: $x^\star + x^\star \cdot y \cdot (x + y)^\star = (x + y)^\star$
⟨*proof*⟩

The following property appears in process algebra.

**lemma** *troeger*: $(x + y)^\star \cdot z = x^\star \cdot (y \cdot (x + y)^\star \cdot z + z)$
⟨*proof*⟩

The following properties are related to a property from propositional dynamic logic which has been attributed to Albert Meyer [18]. Here we prove it as a theorem of Kleene algebra.

**lemma** *star-square*: $(x \cdot x)^\star \leq x^\star$
⟨*proof*⟩

**lemma** *meyer-1* [*simp*]: $(1 + x) \cdot (x \cdot x)^\star = x^\star$
⟨*proof*⟩

The following lemma says that transitive elements are equal to their transitive closure.

**lemma** *tc*: $x \cdot x \leq x \Longrightarrow x^\star \cdot x = x$
⟨*proof*⟩

**lemma** *tc-eq*: $x \cdot x = x \Longrightarrow x^\star \cdot x = x$
⟨*proof*⟩

The next fact has been used by Boffa [6] to axiomatise the equational theory of regular expressions.

**lemma** *boffa-var*: $x \cdot x \leq x \Longrightarrow x^\star = 1 + x$
⟨*proof*⟩

**lemma** *boffa*: $x \cdot x = x \Longrightarrow x^\star = 1 + x$
⟨*proof*⟩

**end**

## 7.4 Left Kleene Algebras with Zero

There are applications where only a left zero is assumed, for instance in the context of total correctness and for demonic refinement algebras [31].

**class** *left-kleene-algebra-zerol* = *left-kleene-algebra* + *dioid-one-zerol*
**begin**

**sublocale** *conway*: *near-conway-base-zerol star*
⟨*proof*⟩

**lemma** *star-zero* [*simp*]: $0^\star = 1$
⟨*proof*⟩

In principle, 1 could therefore be defined from 0 in this setting.

**end**

**class** *left-kleene-algebra-zero* = *left-kleene-algebra-zerol* + *dioid-one-zero*

## 7.5 Pre-Kleene Algebras

Pre-Kleene algebras are essentially probabilistic Kleene algebras [24]. They have a weaker right star unfold axiom. We are still looking for theorems that could be proved in this setting.

**class** *pre-kleene-algebra* = *left-pre-kleene-algebra* +
**assumes** *weak-star-unfoldr*: $z + y \cdot (x + 1) \leq y \Longrightarrow z \cdot x^\star \leq y$

## 7.6 Kleene Algebras

**class** *kleene-algebra-zerol* = *left-kleene-algebra-zerol* +
  **assumes** *star-inductr*: $z + y \cdot x \leq y \Longrightarrow z \cdot x^\star \leq y$

**begin**

**lemma** *star-sim2*: $z \cdot x \leq y \cdot z \Longrightarrow z \cdot x^\star \leq y^\star \cdot z$
⟨*proof*⟩

**sublocale** *conway*: *pre-conway star*
  ⟨*proof*⟩

**lemma** *star-inductr-var*: $y \cdot x \leq y \Longrightarrow y \cdot x^\star \leq y$
  ⟨*proof*⟩

**lemma** *star-inductr-var-equiv*: $y \cdot x \leq y \longleftrightarrow y \cdot x^\star \leq y$
  ⟨*proof*⟩

**lemma** *star-sim3*: $z \cdot x = y \cdot z \Longrightarrow z \cdot x^\star = y^\star \cdot z$
  ⟨*proof*⟩

**lemma** *star-sim4*: $x \cdot y \leq \ y \cdot x \Longrightarrow x^\star \cdot y^\star \leq y^\star \cdot x^\star$
  ⟨*proof*⟩

**lemma** *star-inductr-eq*: $z + y \cdot x = y \Longrightarrow z \cdot x^\star \leq y$
  ⟨*proof*⟩

**lemma** *star-inductr-var-eq*: $y \cdot x = y \Longrightarrow y \cdot x^\star \leq y$
  ⟨*proof*⟩

**lemma** *star-inductr-var-eq2*: $y \cdot x = y \Longrightarrow y \cdot x^\star = y$
  ⟨*proof*⟩

**lemma** *bubble-sort*: $y \cdot x \leq x \cdot y \Longrightarrow (x + y)^\star = x^\star \cdot y^\star$
  ⟨*proof*⟩

**lemma** *independence1*: $x \cdot y = 0 \Longrightarrow x^\star \cdot y = y$
⟨*proof*⟩

**lemma** *independence2*: $x \cdot y = 0 \Longrightarrow x \cdot y^\star = x$
  ⟨*proof*⟩

**lemma** *lazycomm-var*: $y \cdot x \leq x \cdot (x + y)^\star + y \longleftrightarrow y \cdot x^\star \leq x \cdot (x + y)^\star + y$
⟨*proof*⟩

**lemma** *arden-var*: $(\forall\, y\ v.\ y \leq x \cdot y + v \longrightarrow y \leq x^\star \cdot v) \Longrightarrow z = x \cdot z + w \Longrightarrow z = x^\star \cdot w$
  ⟨*proof*⟩

**lemma** $(\forall\, x\ y.\ y \leq x \cdot y \longrightarrow y = 0) \Longrightarrow y \leq x \cdot y + z \Longrightarrow y \leq x^\star \cdot z$
⟨*proof*⟩

**end**

Finally, here come the Kleene algebras à la Kozen [21]. We only prove quasi-identities in this section. Since left Kleene algebras are complete with respect to the equational theory of regular expressions and regular languages, all identities hold already without the right star induction axiom.

**class** *kleene-algebra* = *left-kleene-algebra-zero* +
**assumes** *star-inductr'*: $z + y \cdot x \leq y \Longrightarrow z \cdot x^\star \leq y$
**begin**

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

**sublocale** *conway-zerol*: *conway star* ⟨*proof*⟩

The next lemma shows that opposites of Kleene algebras (i.e., Kleene algebras with the order of multiplication swapped) are again Kleene algebras.

**lemma** *dual-kleene-algebra*:
*class.kleene-algebra* (*op* +) (*op* ⊙) *1 0* (*op* ≤) (*op* <) *star*
⟨*proof*⟩

**end**

We finish with some properties on (multiplicatively) commutative Kleene algebras. A chapter in Conway's book [8] is devoted to this topic.

**class** *commutative-kleene-algebra* = *kleene-algebra* +
**assumes** *mult-comm* [*ac-simps*]: $x \cdot y = y \cdot x$

**begin**

**lemma** *conway-c3* [*simp*]: $(x + y)^\star = x^\star \cdot y^\star$
⟨*proof*⟩

**lemma** *conway-c4*: $(x^\star \cdot y)^\star = 1 + x^\star \cdot y^\star \cdot y$
⟨*proof*⟩

**lemma** *cka-1*: $(x \cdot y)^\star \leq x^\star \cdot y^\star$
⟨*proof*⟩

**lemma** *cka-2* [*simp*]: $x^\star \cdot (x^\star \cdot y)^\star = x^\star \cdot y^\star$
⟨*proof*⟩

**lemma** *conway-c4-var* [*simp*]: $(x^\star \cdot y^\star)^\star = x^\star \cdot y^\star$
⟨*proof*⟩

**lemma** *conway-c2-var*: $(x \cdot y)^\star \cdot x \cdot y \cdot y^\star \leq (x \cdot y)^\star \cdot y^\star$
⟨*proof*⟩

**lemma** *conway-c2* [*simp*]: $(x \cdot y)^\star \cdot (x^\star + y^\star) = x^\star \cdot y^\star$
⟨*proof*⟩

**end**

**end**

# 8 Models of Kleene Algebras

**theory** *Kleene-Algebra-Models*
**imports** *Kleene-Algebra Dioid-Models*
**begin**

We now show that most of the models considered for dioids are also Kleene algebras. Some of the dioid models cannot be expanded, for instance max-plus and min-plus semirings, but we do not formalise this fact. We also currently do not show that formal powerseries and matrices form Kleene algebras.

The interpretation proofs for some of the following models are quite similar. One could, perhaps, abstract out common reasoning in the future.

## 8.1 Preliminary Lemmas

We first prove two induction-style statements for dioids that are useful for establishing the full induction laws. In the future these will live in a theory file on finite sums for Kleene algebras.

**context** *dioid-one-zero*
**begin**

**lemma** *power-inductl*: $z + x \cdot y \leq y \Longrightarrow (x \hat{} n) \cdot z \leq y$
⟨*proof*⟩

**lemma** *power-inductr*: $z + y \cdot x \leq y \Longrightarrow z \cdot (x \hat{} n) \leq y$
⟨*proof*⟩

**end**

## 8.2 The Powerset Kleene Algebra over a Monoid

We now show that the powerset dioid forms a Kleene algebra. The Kleene star is defined as in language theory.

**lemma** *Un-0-Suc*: $(\bigcup n.\ f\ n) = f\ 0 \cup (\bigcup n.\ f\ (Suc\ n))$
⟨*proof*⟩

**instantiation** *set* :: (*monoid-mult*) *kleene-algebra*
**begin**

**definition** *star-def*: $X^\star = (\bigcup n.\ X\ \hat{}\ n)$

**lemma** *star-elim*: $x \in X^\star \longleftrightarrow (\exists k.\ x \in X\ \hat{}\ k)$
⟨*proof*⟩

**lemma** *star-contl*: $X \cdot Y^\star = (\bigcup n.\ X \cdot Y\ \hat{}\ n)$
⟨*proof*⟩

**lemma** *star-contr*: $X^\star \cdot Y = (\bigcup n.\ X\ \hat{}\ n \cdot Y)$
⟨*proof*⟩

**instance**
⟨*proof*⟩

**end**

## 8.3 Language Kleene Algebras

We now specialise this fact to languages.

**interpretation** *lan-kleene-algebra*: *kleene-algebra op* + *op* · *1*::*'a lan 0 op* ⊆ *op* ⊂ *star* ⟨*proof*⟩

## 8.4 Regular Languages

. . . and further to regular languages. For the sake of simplicity we just copy in the axiomatisation of regular expressions by Krauss and Nipkow [23].

**datatype** *'a rexp* =
  *Zero*
| *One*
| *Atom 'a*
| *Plus 'a rexp 'a rexp*
| *Times 'a rexp 'a rexp*
| *Star 'a rexp*

The interpretation map that induces regular languages as the images of regular expressions in the set of languages has also been adapted from there.

**fun** *lang* :: *'a rexp* ⇒ *'a lan* **where**
  *lang Zero* = *0* —
| *lang One* = *1* — []
| *lang* (*Atom a*) = {[*a*]}
| *lang* (*Plus x y*) = *lang x* + *lang y*
| *lang* (*Times x y*) = *lang x* · *lang y*
| *lang* (*Star x*) = (*lang x*)$^\star$

**typedef** *'a reg-lan = range lang :: 'a lan set*
⟨*proof*⟩

**setup-lifting** *type-definition-reg-lan*

**instantiation** *reg-lan* :: (*type*) *kleene-algebra*
**begin**

**lift-definition** *star-reg-lan :: 'a reg-lan ⇒ 'a reg-lan*
**is** *star*
⟨*proof*⟩

**lift-definition** *zero-reg-lan :: 'a reg-lan*
**is** *0*
⟨*proof*⟩

**lift-definition** *one-reg-lan :: 'a reg-lan*
**is** *1*
⟨*proof*⟩

**lift-definition** *less-eq-reg-lan :: 'a reg-lan ⇒ 'a reg-lan ⇒ bool*
**is** *less-eq* ⟨*proof*⟩

**lift-definition** *less-reg-lan :: 'a reg-lan ⇒ 'a reg-lan ⇒ bool*
**is** *less* ⟨*proof*⟩

**lift-definition** *plus-reg-lan :: 'a reg-lan ⇒ 'a reg-lan ⇒ 'a reg-lan*
**is** *plus*
⟨*proof*⟩

**lift-definition** *times-reg-lan :: 'a reg-lan ⇒ 'a reg-lan ⇒ 'a reg-lan*
**is** *times*
⟨*proof*⟩

**instance**
⟨*proof*⟩

**end**

**interpretation** *reg-lan-kleene-algebra*: *kleene-algebra op + op · 1::'a reg-lan 0 op ≤ op < star* ⟨*proof*⟩

## 8.5 Relation Kleene Algebras

We now show that binary relations form Kleene algebras. While we could have used the reflexive transitive closure operation as the Kleene star, we prefer the equivalent definition of the star as the sum of powers. This essentially allows us to copy previous proofs.

**lemma** *power-is-relpow*: *rel-dioid.power X n* = *X* ^^ *n*
⟨*proof*⟩

**lemma** *rel-star-def*: *X*^* = (⋃ *n*. *rel-dioid.power X n*)
⟨*proof*⟩

**lemma** *rel-star-contl*: *X O Y*^* = (⋃ *n*. *X O rel-dioid.power Y n*)
⟨*proof*⟩

**lemma** *rel-star-contr*: *X*^* *O Y* = (⋃ *n*. (*rel-dioid.power X n*) *O Y*)
⟨*proof*⟩

**interpretation** *rel-kleene-algebra*: *kleene-algebra op* ∪ *op O Id* {} *op* ⊆ *op* ⊂ *rtrancl*
⟨*proof*⟩

## 8.6 Trace Kleene Algebras

Again, the proof that sets of traces form Kleene algebras follows the same schema.

**definition** *t-star* :: ('*p*, '*a*) *trace set* ⇒ ('*p*, '*a*) *trace set* **where**
*t-star X* ≡ ⋃ *n*. *trace-dioid.power X n*

**lemma** *t-star-elim*: *x* ∈ *t-star X* ⟷ (∃ *n*. *x* ∈ *trace-dioid.power X n*)
⟨*proof*⟩

**lemma** *t-star-contl*: *t-prod X* (*t-star Y*) = (⋃ *n*. *t-prod X* (*trace-dioid.power Y n*))
⟨*proof*⟩

**lemma** *t-star-contr*: *t-prod* (*t-star X*) *Y* = (⋃ *n*. *t-prod* (*trace-dioid.power X n*) *Y*)
⟨*proof*⟩

**interpretation** *trace-kleene-algebra*: *kleene-algebra op* ∪ *t-prod t-one t-zero op* ⊆ *op* ⊂ *t-star*
⟨*proof*⟩

## 8.7 Path Kleene Algebras

We start with paths that include the empty path.

**definition** *p-star* :: '*a path set* ⇒ '*a path set* **where**
*p-star X* ≡ ⋃ *n*. *path-dioid.power X n*

**lemma** *p-star-elim*: *x* ∈ *p-star X* ⟷ (∃ *n*. *x* ∈ *path-dioid.power X n*)
⟨*proof*⟩

**lemma** *p-star-contl*: *p-prod X* (*p-star Y*) = (⋃ *n*. *p-prod X* (*path-dioid.power Y n*))
⟨*proof*⟩

**lemma** *p-star-contr*: *p-prod* (*p-star X*) *Y* = (⋃ *n*. *p-prod* (*path-dioid.power X n*) *Y*)
⟨*proof*⟩

**interpretation** *path-kleene-algebra*: *kleene-algebra op* ∪ *p-prod p-one* {} *op* ⊆ *op* ⊂ *p-star*
⟨*proof*⟩

We now consider a notion of paths that does not include the empty path.

**definition** *pp-star* :: *'a ppath set* ⇒ *'a ppath set* **where**
*pp-star X* ≡ ⋃ *n*. *ppath-dioid.power X n*

**lemma** *pp-star-elim*: *x* ∈ *pp-star X* ⟷ (∃ *n*. *x* ∈ *ppath-dioid.power X n*)
⟨*proof*⟩

**lemma** *pp-star-contl*: *pp-prod X* (*pp-star Y*) = (⋃ *n*. *pp-prod X* (*ppath-dioid.power Y n*))
⟨*proof*⟩

**lemma** *pp-star-contr*: *pp-prod* (*pp-star X*) *Y* = (⋃ *n*. *pp-prod* (*ppath-dioid.power X n*) *Y*)
⟨*proof*⟩

**interpretation** *ppath-kleene-algebra*: *kleene-algebra op* ∪ *pp-prod pp-one* {} *op* ⊆ *op* ⊂ *pp-star*
⟨*proof*⟩

## 8.8 The Distributive Lattice Kleene Algebra

In the case of bounded distributive lattices, the star maps all elements to to the maximal element.

**definition** (**in** *bounded-distributive-lattice*) *bdl-star* :: *'a* ⇒ *'a* **where**
*bdl-star x* = *top*

**sublocale** *bounded-distributive-lattice* ⊆ *kleene-algebra sup inf top bot less-eq less bdl-star*
⟨*proof*⟩

## 8.9 The Min-Plus Kleene Algebra

One cannot define a Kleene star for max-plus and min-plus algebras that range over the real numbers. Here we define the star for a min-plus algebra restricted to natural numbers and $+\infty$. The resulting Kleene algebra is

commutative. Similar variants can be obtained for max-plus algebras and other algebras ranging over the positive or negative integers.

**instantiation** *pnat* :: *commutative-kleene-algebra*
**begin**

**definition** *star-pnat* **where**
$x^\star \equiv (1{::}pnat)$

**instance**
⟨*proof*⟩

**end**

**end**

# 9 Omega Algebras

**theory** *Omega-Algebra*
**imports** *Kleene-Algebra*
**begin**

*Omega algebras* [7] extend Kleene algebras by an $\omega$-operation that axiomatizes infinite iteration (just like the Kleene star axiomatizes finite iteration).

## 9.1 Left Omega Algebras

In this section we consider *left omega algebras*, i.e., omega algebras based on left Kleene algebras. Surprisingly, we are still looking for statements mentioning $\omega$ that are true in omega algebras, but do not already hold in left omega algebras.

**class** *left-omega-algebra* = *left-kleene-algebra-zero* + *omega-op* +
**assumes** *omega-unfold*: $x^\omega \leq x \cdot x^\omega$
**and** *omega-coinduct*: $y \leq z + x \cdot y \Longrightarrow y \leq x^\omega + x^\star \cdot z$
**begin**

First we prove some variants of the coinduction axiom.

**lemma** *omega-coinduct-var1*: $y \leq 1 + x \cdot y \Longrightarrow y \leq x^\omega + x^\star$
⟨*proof*⟩

**lemma** *omega-coinduct-var2*: $y \leq x \cdot y \Longrightarrow y \leq x^\omega$
⟨*proof*⟩

**lemma** *omega-coinduct-eq*: $y = z + x \cdot y \Longrightarrow y \leq x^\omega + x^\star \cdot z$
⟨*proof*⟩

**lemma** *omega-coinduct-eq-var1*: $y = 1 + x \cdot y \Longrightarrow y \leq x^\omega + x^\star$
⟨*proof*⟩

**lemma** *omega-coinduct-eq-var2*: $y = x \cdot y \Longrightarrow y \leq x^\omega$
⟨*proof*⟩

**lemma** $y = x \cdot y + z \Longrightarrow y = x^\star \cdot z + x^\omega$

⟨*proof*⟩

**lemma** $y = 1 + x \cdot y \Longrightarrow y = x^\omega + x^\star$

⟨*proof*⟩

**lemma** $y = x \cdot y \Longrightarrow y = x^\omega$

⟨*proof*⟩

Next we strengthen the unfold law to an equation.

**lemma** *omega-unfold-eq* [*simp*]: $x \cdot x^\omega = x^\omega$
⟨*proof*⟩

**lemma** *omega-unfold-var*: $z + x \cdot x^\omega \leq x^\omega + x^\star \cdot z$
⟨*proof*⟩

**lemma** $z + x \cdot x^\omega = x^\omega + x^\star \cdot z$

⟨*proof*⟩

We now prove subdistributivity and isotonicity of omega.

**lemma** *omega-subdist*: $x^\omega \leq (x + y)^\omega$
⟨*proof*⟩

**lemma** *omega-iso*: $x \leq y \Longrightarrow x^\omega \leq y^\omega$
⟨*proof*⟩

**lemma** *omega-subdist-var*: $x^\omega + y^\omega \leq (x + y)^\omega$
⟨*proof*⟩

**lemma** *zero-omega* [*simp*]: $0^\omega = 0$
⟨*proof*⟩

The next lemma is another variant of omega unfold

**lemma** *star-omega-1* [*simp*]: $x^\star \cdot x^\omega = x^\omega$
⟨*proof*⟩

The next lemma says that $(1::'a)^\omega$ is the maximal element of omega algebra. We therefore baptise it $\top$.

**lemma** *max-element*: $x \leq 1^\omega$
⟨*proof*⟩

**definition** *top* ($\top$)
**where** $\top = 1^\omega$

**lemma** *star-omega-3* [*simp*]: $(x^\star)^\omega = \top$
⟨*proof*⟩

The following lemma is strange since it is counterintuitive that one should be able to append something after an infinite iteration.

**lemma** *omega-1*: $x^\omega \cdot y \leq x^\omega$
⟨*proof*⟩

**lemma** $x^\omega \cdot y = x^\omega$

⟨*proof*⟩

**lemma** *omega-sup-id*: $1 \leq y \Longrightarrow x^\omega \cdot y = x^\omega$
⟨*proof*⟩

**lemma** *omega-top* [*simp*]: $x^\omega \cdot \top = x^\omega$
⟨*proof*⟩

**lemma** *supid-omega*: $1 \leq x \Longrightarrow x^\omega = \top$
⟨*proof*⟩

**lemma** $x^\omega = \top \Longrightarrow 1 \leq x$

⟨*proof*⟩

Next we prove a simulation law for the omega operation

**lemma** *omega-simulation*: $z \cdot x \leq y \cdot z \Longrightarrow z \cdot x^\omega \leq y^\omega$
⟨*proof*⟩

**lemma** $z \cdot x \leq y \cdot z \Longrightarrow z \cdot x^\omega \leq y^\omega \cdot z$

⟨*proof*⟩

**lemma** $y \cdot z \leq z \cdot x \Longrightarrow y^\omega \leq z \cdot x^\omega$

⟨*proof*⟩

**lemma** $y \cdot z \leq z \cdot x \Longrightarrow y^\omega \cdot z \leq x^\omega$

⟨*proof*⟩

Next we prove transitivity of omega elements.

**lemma** *omega-omega*: $(x^\omega)^\omega \leq x^\omega$
⟨*proof*⟩

The next lemmas are axioms of Wagner's complete axiomatisation for omega-regular languages [32], but in a slightly different setting.

**lemma** *wagner-1* [*simp*]: $(x \cdot x^\star)^\omega = x^\omega$
⟨*proof*⟩

**lemma** *wagner-2-var*: $x \cdot (y \cdot x)^\omega \leq (x \cdot y)^\omega$
⟨*proof*⟩

**lemma** *wagner-2* [*simp*]: $x \cdot (y \cdot x)^\omega = (x \cdot y)^\omega$
⟨*proof*⟩

This identity is called (A8) in Wagner's paper.

**lemma** *wagner-3*:
**assumes** $x \cdot (x + y)^\omega + z = (x + y)^\omega$
**shows** $(x + y)^\omega = x^\omega + x^\star \cdot z$
⟨*proof*⟩

This identity is called (R4) in Wagner's paper.

**lemma** *wagner-1-var* [*simp*]: $(x^\star \cdot x)^\omega = x^\omega$
⟨*proof*⟩

**lemma** *star-omega-4* [*simp*]: $(x^\omega)^\star = 1 + x^\omega$
⟨*proof*⟩

**lemma** *star-omega-5* [*simp*]: $x^\omega \cdot (x^\omega)^\star = x^\omega$
⟨*proof*⟩

The next law shows how omegas below a sum can be unfolded.

**lemma** *omega-sum-unfold*: $x^\omega + x^\star \cdot y \cdot (x + y)^\omega = (x + y)^\omega$
⟨*proof*⟩

The next two lemmas apply induction and coinduction to this law.

**lemma** *omega-sum-unfold-coind*: $(x + y)^\omega \leq (x^\star \cdot y)^\omega + (x^\star \cdot y)^\star \cdot x^\omega$
⟨*proof*⟩

**lemma** *omega-sum-unfold-ind*: $(x^\star \cdot y)^\star \cdot x^\omega \leq (x + y)^\omega$
⟨*proof*⟩

**lemma** *wagner-1-gen*: $(x \cdot y^\star)^\omega \leq (x + y)^\omega$
⟨*proof*⟩

**lemma** *wagner-1-var-gen*: $(x^\star \cdot y)^\omega \leq (x + y)^\omega$
⟨*proof*⟩

The next lemma is a variant of the denest law for the star at the level of omega.

**lemma** *omega-denest* [*simp*]: $(x + y)^\omega = (x^\star \cdot y)^\omega + (x^\star \cdot y)^\star \cdot x^\omega$

⟨*proof*⟩

The next lemma yields a separation theorem for infinite iteration in the presence of a quasicommutation property. A nondeterministic loop over $x$ and $y$ can be refined into separate infinite loops over $x$ and $y$.

**lemma** *omega-sum-refine*:
**assumes** $y \cdot x \leq x \cdot (x + y)^\star$
**shows** $(x + y)^\omega = x^\omega + x^\star \cdot y^\omega$
⟨*proof*⟩

The following theorem by Bachmair and Dershowitz [4] is a corollary.

**lemma** *bachmair-dershowitz*:
**assumes** $y \cdot x \leq x \cdot (x + y)^\star$
**shows** $(x + y)^\omega = 0 \longleftrightarrow x^\omega + y^\omega = 0$
⟨*proof*⟩

The next lemmas consider an abstract variant of the empty word property from language theory and match it with the absence of infinite iteration [28].

**definition** (**in** *dioid-one-zero*) *ewp*
**where** $ewp\ x \equiv \neg(\forall\, y.\ y \leq x \cdot y \longrightarrow y = 0)$

**lemma** *ewp-super-id1*: $0 \neq 1 \Longrightarrow 1 \leq x \Longrightarrow ewp\ x$
⟨*proof*⟩

**lemma** $0 \neq 1 \Longrightarrow 1 \leq x \longleftrightarrow ewp\ x$

⟨*proof*⟩

The next facts relate the absence of the empty word property with the absence of infinite iteration.

**lemma** *ewp-neg-and-omega*: $\neg\ ewp\ x \longleftrightarrow x^\omega = 0$
⟨*proof*⟩

**lemma** *ewp-alt1*: $(\forall\, z.\ x^\omega \leq x^\star \cdot z) \longleftrightarrow (\forall\, y\ z.\ y \leq x \cdot y + z \longrightarrow y \leq x^\star \cdot z)$
⟨*proof*⟩

**lemma** *ewp-alt*: $x^\omega = 0 \longleftrightarrow (\forall\, y\ z.\ y \leq x \cdot y + z \longrightarrow y \leq x^\star \cdot z)$
⟨*proof*⟩

So we have obtained a condition for Arden's lemma in omega algebra.

**lemma** *omega-super-id1*: $0 \neq 1 \Longrightarrow 1 \leq x \Longrightarrow x^\omega \neq 0$
⟨*proof*⟩

**lemma** *omega-super-id2*: $0 \neq 1 \Longrightarrow x^\omega = 0 \Longrightarrow \neg(1 \leq x)$
⟨*proof*⟩

The next lemmas are abstract versions of Arden's lemma from language theory.

**lemma** *ardens-lemma-var*:
**assumes** $x^\omega = 0$
**and** $z + x \cdot y = y$
**shows** $x^\star \cdot z = y$
⟨*proof*⟩

**lemma** *ardens-lemma*: $\neg$ *ewp* $x \Longrightarrow z + x \cdot y = y \Longrightarrow x^\star \cdot z = y$
⟨*proof*⟩

**lemma** *ardens-lemma-equiv*:
**assumes** $\neg$ *ewp* $x$
**shows** $z + x \cdot y = y \longleftrightarrow x^\star \cdot z = y$
⟨*proof*⟩

**lemma** *ardens-lemma-var-equiv*: $x^\omega = 0 \Longrightarrow (z + x \cdot y = y \longleftrightarrow x^\star \cdot z = y)$
⟨*proof*⟩

**lemma** *arden-conv1*: $(\forall\, y\ z.\ z + x \cdot y = y \longrightarrow x^\star \cdot z = y) \Longrightarrow \neg$ *ewp* $x$
⟨*proof*⟩

**lemma** *arden-conv2*: $(\forall\, y\ z.\ z + x \cdot y = y \longrightarrow x^\star \cdot z = y) \Longrightarrow x^\omega = 0$
⟨*proof*⟩

**lemma** *arden-var3*: $(\forall\, y\ z.\ z + x \cdot y = y \longrightarrow x^\star \cdot z = y) \longleftrightarrow x^\omega = 0$
⟨*proof*⟩

**end**

### 9.2 Omega Algebras

**class** *omega-algebra* = *kleene-algebra* + *left-omega-algebra*

**end**

## 10 Models of Omega Algebras

**theory** *Omega-Algebra-Models*
**imports** *Omega-Algebra Kleene-Algebra-Models*
**begin**

The trace, path and language model are not really interesting in this setting.

### 10.1 Relation Omega Algebras

In the relational model, the omega of a relation relates all those elements in the domain of the relation, from which an infinite chain starts, with all other elements; all other elements are not related to anything [19]. Thus, the omega of a relation is most naturally defined coinductively.

**coinductive-set** *omega* :: $('a \times 'a)$ *set* $\Rightarrow$ $('a \times 'a)$ *set* **for** $R$ **where**
$[\![ (x, y) \in R; (y, z) \in omega\ R ]\!] \Longrightarrow (x, z) \in omega\ R$

Isabelle automatically derives a case rule and a coinduction theorem for *Omega-Algebra-Models.omega*. We prove slightly more elegant variants.

**lemma** *omega-cases*: $(x, z) \in omega\ R \Longrightarrow$
$(\bigwedge y.\ (x, y) \in R \Longrightarrow (y, z) \in omega\ R \Longrightarrow P) \Longrightarrow P$
⟨*proof*⟩

**lemma** *omega-coinduct*: $X\ x\ z \Longrightarrow$
$(\bigwedge x\ z.\ X\ x\ z \Longrightarrow \exists y.\ (x, y) \in R \wedge (X\ y\ z \vee (y, z) \in omega\ R)) \Longrightarrow$
$(x, z) \in omega\ R$
⟨*proof*⟩

**lemma** *omega-weak-coinduct*: $X\ x\ z \Longrightarrow$
$(\bigwedge x\ z.\ X\ x\ z \Longrightarrow \exists y.\ (x, y) \in R \wedge X\ y\ z) \Longrightarrow$
$(x, z) \in omega\ R$
⟨*proof*⟩

**interpretation** *rel-omega-algebra*: *omega-algebra op* $\cup$ *op O Id* $\{\}$ *op* $\subseteq$ *op* $\subset$ *rtrancl omega*
⟨*proof*⟩

**end**

# 11 Demonic Refinement Algebras

**theory** *DRA*
**imports** *Kleene-Algebra*
**begin**

A demonic refinement algebra *DRA) [31] is a Kleene algebra without right annihilation plus an operation for possibly infinite iteration.

**class** *dra* = *kleene-algebra-zerol* +
**fixes** *strong-iteration* :: $'a \Rightarrow 'a$ $(\text{-}^{\infty}\ [101]\ 100)$
**assumes** *iteration-unfoldl* [*simp*] : $1 + x \cdot x^{\infty} = x^{\infty}$
**and** *coinduction*: $y \leq z + x \cdot y \longrightarrow y \leq x^{\infty} \cdot z$
**and** *isolation* [*simp*]: $x^{\star} + x^{\infty} \cdot 0 = x^{\infty}$
**begin**

$\top$ is an abort statement, defined as an infinite skip. It is the maximal element of any DRA.

**abbreviation** *top-elem* :: $'a$ $(\top)$ **where** $\top \equiv 1^{\infty}$

Simple/basic lemmas about the iteration operator

**lemma** *iteration-refl*: $1 \leq x^{\infty}$
⟨*proof*⟩

**lemma** *iteration-1l*: $x \cdot x^\infty \leq x^\infty$
⟨*proof*⟩

**lemma** *top-ref*: $x \leq \top$
⟨*proof*⟩

**lemma** *it-ext*: $x \leq x^\infty$
⟨*proof*⟩

**lemma** *it-idem* [*simp*]: $(x^\infty)^\infty = x^\infty$

⟨*proof*⟩

**lemma** *top-mult-annil* [*simp*]: $\top \cdot x = \top$
⟨*proof*⟩

**lemma** *top-add-annil* [*simp*]: $\top + x = \top$
⟨*proof*⟩

**lemma** *top-elim*: $x \cdot y \leq x \cdot \top$
⟨*proof*⟩

**lemma** *iteration-unfoldl-distl* [*simp*]: $y + y \cdot x \cdot x^\infty = y \cdot x^\infty$
⟨*proof*⟩

**lemma** *iteration-unfoldl-distr* [*simp*]: $y + x \cdot x^\infty \cdot y = x^\infty \cdot y$
⟨*proof*⟩

**lemma** *iteration-unfoldl'* [*simp*]: $z \cdot y + z \cdot x \cdot x^\infty \cdot y = z \cdot x^\infty \cdot y$
⟨*proof*⟩

**lemma** *iteration-idem* [*simp*]: $x^\infty \cdot x^\infty = x^\infty$
⟨*proof*⟩

**lemma** *iteration-induct*: $x \cdot x^\infty \leq x^\infty \cdot x$
⟨*proof*⟩

**lemma** *iteration-ref-star*: $x^\star \leq x^\infty$
⟨*proof*⟩

**lemma** *iteration-subdist*: $x^\infty \leq (x + y)^\infty$
⟨*proof*⟩

**lemma** *iteration-iso*: $x \leq y \Longrightarrow x^\infty \leq y^\infty$
⟨*proof*⟩

**lemma** *iteration-unfoldr* [*simp*]: $1 + x^\infty \cdot x = x^\infty$
⟨*proof*⟩

**lemma** *iteration-unfoldr-distl* [*simp*]: $y + y \cdot x^\infty \cdot x = y \cdot x^\infty$
⟨*proof*⟩

**lemma** *iteration-unfoldr-distr* [*simp*]: $y + x^\infty \cdot x \cdot y = x^\infty \cdot y$
⟨*proof*⟩

**lemma** *iteration-unfold-eq*: $x^\infty \cdot x = x \cdot x^\infty$
⟨*proof*⟩

**lemma** *iteration-unfoldr'* [*simp*]: $z \cdot y + z \cdot x^\infty \cdot x \cdot y = z \cdot x^\infty \cdot y$
⟨*proof*⟩

**lemma** *iteration-double* [*simp*]: $(x^\infty)^\infty = \top$
⟨*proof*⟩

**lemma** *star-iteration* [*simp*]: $(x^\star)^\infty = \top$
⟨*proof*⟩

**lemma** *iteration-star* [*simp*]: $(x^\infty)^\star = x^\infty$
⟨*proof*⟩

**lemma** *iteration-star2* [*simp*]: $x^\star \cdot x^\infty = x^\infty$
⟨*proof*⟩

**lemma** *iteration-zero* [*simp*]: $0^\infty = 1$
⟨*proof*⟩

**lemma** *iteration-annil* [*simp*]: $(x \cdot 0)^\infty = 1 + x \cdot 0$
⟨*proof*⟩

**lemma** *iteration-subdenest*: $x^\infty \cdot y^\infty \leq (x + y)^\infty$
⟨*proof*⟩

**lemma** *sup-id-top*: $1 \leq y \Longrightarrow y \cdot \top = \top$
⟨*proof*⟩

**lemma** *iteration-top* [*simp*]: $x^\infty \cdot \top = \top$
⟨*proof*⟩

Next, we prove some simulation laws for data refinement.

**lemma** *iteration-sim*: $z \cdot y \leq x \cdot z \Longrightarrow z \cdot y^\infty \leq x^\infty \cdot z$
⟨*proof*⟩

Nitpick gives a counterexample to the dual simulation law.

**lemma** $y \cdot z \leq z \cdot x \Longrightarrow y^\infty \cdot z \leq z \cdot x^\infty$

⟨*proof*⟩

Next, we prove some sliding laws.

**lemma** *iteration-slide-var*: $x \cdot (y \cdot x)^\infty \leq (x \cdot y)^\infty \cdot x$
⟨*proof*⟩

**lemma** *iteration-prod-unfold* [*simp*]: $1 + y \cdot (x \cdot y)^\infty \cdot x = (y \cdot x)^\infty$
⟨*proof*⟩

**lemma** *iteration-slide*: $x \cdot (y \cdot x)^\infty = (x \cdot y)^\infty \cdot x$
⟨*proof*⟩

**lemma** *star-iteration-slide* [*simp*]: $y^\star \cdot (x^\star \cdot y)^\infty = (x^\star \cdot y)^\infty$
⟨*proof*⟩

The following laws are called denesting laws.

**lemma** *iteration-sub-denest*: $(x + y)^\infty \leq x^\infty \cdot (y \cdot x^\infty)^\infty$
⟨*proof*⟩

**lemma** *iteration-denest*: $(x + y)^\infty = x^\infty \cdot (y \cdot x^\infty)^\infty$
⟨*proof*⟩

**lemma** *iteration-denest2* [*simp*]: $y^\star \cdot x \cdot (x + y)^\infty + y^\infty = (x + y)^\infty$
⟨*proof*⟩

**lemma** *iteration-denest3*: $(y^\star \cdot x)^\infty \cdot y^\infty = (x + y)^\infty$
⟨*proof*⟩

Now we prove separation laws for reasoning about distributed systems in the context of action systems.

**lemma** *iteration-sep*: $y \cdot x \leq x \cdot y \Longrightarrow (x + y)^\infty = x^\infty \cdot y^\infty$
⟨*proof*⟩

**lemma** *iteration-sim2*: $y \cdot x \leq x \cdot y \Longrightarrow y^\infty \cdot x^\infty \leq x^\infty \cdot y^\infty$
⟨*proof*⟩

**lemma** *iteration-sep2*: $y \cdot x \leq x \cdot y^\star \Longrightarrow (x + y)^\infty = x^\infty \cdot y^\infty$
⟨*proof*⟩

**lemma** *iteration-sep3*: $y \cdot x \leq x \cdot (x + y) \Longrightarrow (x + y)^\infty = x^\infty \cdot y^\infty$
⟨*proof*⟩

**lemma** *iteration-sep4*: $y \cdot 0 = 0 \Longrightarrow z \cdot x = 0 \Longrightarrow y \cdot x \leq (x + z) \cdot y^\star \Longrightarrow (x + y + z)^\infty = x^\infty \cdot (y + z)^\infty$
⟨*proof*⟩

Finally, we prove some blocking laws.

Nitpick refutes the next lemma.

**lemma** $x \cdot y = 0 \Longrightarrow x^\infty \cdot y = y$

⟨*proof*⟩

**lemma** *iteration-idep*: $x \cdot y = 0 \Longrightarrow x \cdot y^\infty = x$
⟨*proof*⟩

Nitpick refutes the next lemma.

**lemma** $y \cdot w \leq x \cdot y + z \Longrightarrow y \cdot w^\infty \leq x^\infty \cdot z$

⟨*proof*⟩

At the end of this file, we consider a data refinement example from von Wright [30].

**lemma** *data-refinement*:
**assumes** $s' \leq s \cdot z$ **and** $z \cdot e' \leq e$ **and** $z \cdot a' \leq a \cdot z$ **and** $z \cdot b \leq z$ **and** $b^\infty = b^\star$
**shows** $s' \cdot (a' + b)^\infty \cdot e' \leq s \cdot a^\infty \cdot e$
⟨*proof*⟩

**end**

**end**

# 12 Propositional Hoare Logic for Conway and Kleene Algebra

**theory** *PHL-KA*
**imports** *Kleene-Algebra*

**begin**

This is a minimalist Hoare logic developed in the context of pre-dioids. In near-dioids, the sequencing rule would not be derivable. Iteration is modelled by a function that needs to satisfy a simulation law.

The main assumtions on pre-dioid elements needed to derive the Hoare rules are preservation properties; an additional distributivity propery is needed for the conditional rule.

This Hoare logic can be instantated in various ways. It covers notions of finite and possibly infinite iteration. In this theory, it it specialised to Conway and Kleene algebras.

**class** *it-pre-dioid* = *pre-dioid-one* +
**fixes** *it* :: $'a \Rightarrow 'a$
**assumes** *it-simr*: $y \cdot x \leq x \cdot y \Longrightarrow y \cdot it\ x \leq it\ x \cdot y$

**begin**

**lemma** *phl-while*:
**assumes** $p \cdot s \leq s \cdot p \cdot s$ **and** $p \cdot w \leq w \cdot p \cdot w$

**and** $(p \cdot s) \cdot x \leq x \cdot p$
**shows** $p \cdot (it\ (s \cdot x) \cdot w) \ \leq it\ (s \cdot x) \cdot w \cdot (p \cdot w)$
⟨*proof*⟩

**end**

Next we define a Hoare triple to make the format of the rules more explicit.

**context** *pre-dioid-one*
**begin**

**abbreviation** (**in** *near-dioid*) *ht* :: $'a \Rightarrow 'a \Rightarrow 'a \Rightarrow bool$ ($\{|$-$|\}$-$\{|$-$|\}$) **where**
$\{|x|\}\ y\ \{|z|\} \equiv x \cdot y \leq y \cdot z$

**lemma** *ht-phl-skip*: $\{|x|\}\ 1\ \{|x|\}$
⟨*proof*⟩

**lemma** *ht-phl-cons1*: $x \leq w \Longrightarrow \{|w|\}\ y\ \{|z|\} \Longrightarrow \{|x|\}\ y\ \{|z|\}$
⟨*proof*⟩

**lemma** *ht-phl-cons2*: $w \leq x \Longrightarrow \{|z|\}\ y\ \{|w|\} \Longrightarrow \{|z|\}\ y\ \{|x|\}$
⟨*proof*⟩

**lemma** *ht-phl-seq*: $\{|p|\}\ x\ \{|r|\} \Longrightarrow \{|r|\}\ y\ \{|q|\} \Longrightarrow \{|p|\}\ x \cdot y\ \{|q|\}$
⟨*proof*⟩

**lemma** *ht-phl-cond*:
**assumes** $u \cdot v \leq v \cdot u \cdot v$ **and** $u \cdot w \leq w \cdot u \cdot w$
**and** $\bigwedge x\ y.\ u \cdot (x + y) \leq u \cdot x + u \cdot y$
**and** $\{|u \cdot v|\}\ x\ \{|z|\}$ **and** $\{|u \cdot w|\}\ y\ \{|z|\}$
**shows** $\{|u|\}\ (v \cdot x + w \cdot y)\ \{|z|\}$
⟨*proof*⟩

**lemma** *ht-phl-export1*:
**assumes** $x \cdot y \leq y \cdot x \cdot y$
**and** $\{|x \cdot y|\}\ z\ \{|w|\}$
**shows** $\{|x|\}\ y \cdot z\ \{|w|\}$
⟨*proof*⟩

**lemma** *ht-phl-export2*:
**assumes** $z \cdot w \leq w \cdot z \cdot w$
**and** $\{|x|\}\ y\ \{|z|\}$
**shows** $\{|x|\}\ y \cdot w\ \{|z \cdot w|\}$
⟨*proof*⟩

**end**

**context** *it-pre-dioid* **begin**

**lemma** *ht-phl-while*:

**assumes** $p \cdot s \leq s \cdot p \cdot s$ **and** $p \cdot w \leq w \cdot p \cdot w$
**and** $\{\!|p \cdot s|\!\}\ x\ \{\!|p|\!\}$
**shows** $\{\!|p|\!\}\ it\ (s \cdot x) \cdot w\ \{\!|p \cdot w|\!\}$
⟨*proof*⟩

**end**

**sublocale** *pre-conway* < *phl*: *it-pre-dioid* **where** *it* = *dagger*
⟨*proof*⟩

**sublocale** *kleene-algebra* < *phl*: *it-pre-dioid* **where** *it* = *star* ⟨*proof*⟩

**end**

# 13 Propositional Hoare Logic for Demonic Refinement Algebra

In this section the generic iteration operator is instantiated to the strong iteration operator of demonic refinement algebra that models possibly infinite iteration.

**theory** *PHL-DRA*
**imports** *DRA PHL-KA*
**begin**

**sublocale** *dra* < *total-phl*: *it-pre-dioid* **where** *it* = *strong-iteration*
⟨*proof*⟩

**end**

# 14 Finite Suprema

**theory** *Finite-Suprema*
**imports** *Dioid*
**begin**

This file contains an adaptation of Isabelle's library for finite sums to the case of (join) semilattices and dioids. In this setting, addition is idempotent; finite sums are finite suprema.

We add some basic properties of finite suprema for (join) semilattices and dioids.

## 14.1 Auxiliary Lemmas

**lemma** *fun-im*: $\{f\ a\ |a.\ a \in A\} = \{b.\ b \in f\ `\ A\}$
⟨*proof*⟩

**lemma** *fset-to-im*: $\{f\ x\ |x.\ x \in X\} = f\ `\ X$
⟨*proof*⟩

**lemma** *cart-flip-aux*: $\{f\ (snd\ p)\ (fst\ p)\ |p.\ p \in (B \times A)\} = \{f\ (fst\ p)\ (snd\ p)\ |p.\ p \in (A \times B)\}$
⟨*proof*⟩

**lemma** *cart-flip*: $(\lambda p.\ f\ (snd\ p)\ (fst\ p))\ `\ (B \times A) = (\lambda p.\ f\ (fst\ p)\ (snd\ p))\ `\ (A \times B)$
⟨*proof*⟩

**lemma** *fprod-aux*: $\{x \cdot y\ |x\ y.\ x \in (f\ `\ A) \wedge y \in (g\ `\ B)\} = \{f\ x \cdot g\ y\ |x\ y.\ x \in A \wedge y \in B\}$
⟨*proof*⟩

## 14.2 Finite Suprema in Semilattices

The first lemma shows that, in the context of semilattices, finite sums satisfy the defining property of finite suprema.

**lemma** *sum-sup*:
**assumes** *finite* $(A :: 'a::join\text{-}semilattice\text{-}zero\ set)$
**shows** $\sum A \leq z \longleftrightarrow (\forall a \in A.\ a \leq z)$
⟨*proof*⟩

This immediately implies some variants.

**lemma** *sum-less-eqI*:
$(\bigwedge x.\ x \in A \Longrightarrow f\ x \leq y) \Longrightarrow sum\ f\ A \leq (y::'a::join\text{-}semilattice\text{-}zero)$
⟨*proof*⟩

**lemma** *sum-less-eqE*:
$[\![\ sum\ f\ A \leq y;\ x \in A;\ finite\ A\ ]\!] \Longrightarrow f\ x \leq (y::'a::join\text{-}semilattice\text{-}zero)$
⟨*proof*⟩

**lemma** *sum-fun-image-sup*:
**fixes** $f :: 'a \Rightarrow 'b::join\text{-}semilattice\text{-}zero$
**assumes** *finite* $(A :: 'a\ set)$
**shows** $\sum (f\ `\ A) \leq z \longleftrightarrow (\forall a \in A.\ f\ a \leq z)$
⟨*proof*⟩

**lemma** *sum-fun-sup*:
**fixes** $f :: 'a \Rightarrow 'b::join\text{-}semilattice\text{-}zero$
**assumes** *finite* $(A ::'a\ set)$
**shows** $\sum \{f\ a \mid a.\ a \in A\} \leq z \longleftrightarrow (\forall a \in A.\ f\ a \leq z)$
⟨*proof*⟩

**lemma** *sum-intro*:
**assumes** *finite* $(A :: 'a::join\text{-}semilattice\text{-}zero\ set)$ **and** *finite* $B$
**shows** $(\forall a \in A.\ \exists b \in B.\ a \leq b) \longrightarrow (\sum A \leq \sum B)$

⟨*proof*⟩

Next we prove an additivity property for suprema.

**lemma** *sum-union*:
**assumes** *finite* (*A* :: *'a*::*join-semilattice-zero set*)
**and** *finite* (*B* :: *'a*::*join-semilattice-zero set*)
**shows** $\sum (A \cup B) = \sum A + \sum B$
⟨*proof*⟩

It follows that the sum (supremum) of a two-element set is the join of its elements.

**lemma** *sum-bin*[*simp*]: $\sum \{(x :: 'a::join\text{-}semilattice\text{-}zero), y\} = x + y$
⟨*proof*⟩

Next we show that finite suprema are order preserving.

**lemma** *sum-iso*:
**assumes** *finite* (*B* :: *'a*::*join-semilattice-zero set*)
**shows** $A \subseteq B \longrightarrow \sum\ A \leq \sum\ B$
⟨*proof*⟩

The following lemmas state unfold properties for suprema and finite sets. They are subtly different from the non-idempotent case, where additional side conditions are required.

**lemma** *sum-insert* [*simp*]:
**assumes** *finite* (*A* :: *'a*::*join-semilattice-zero set*)
**shows** $\sum (insert\ x\ A) = x + \sum A$
⟨*proof*⟩

**lemma** *sum-fun-insert*:
**fixes** *f* :: *'a* ⇒ *'b*::*join-semilattice-zero*
**assumes** *finite* (*A* :: *'a set*)
**shows** $\sum (f\ `\ (insert\ x\ A)) = f\ x + \sum (f\ `\ A)$
⟨*proof*⟩

Now we show that set comprehensions with nested suprema can be flattened.

**lemma** *flatten1-im*:
**fixes** *f* :: *'a* ⇒ *'a* ⇒ *'b*::*join-semilattice-zero*
**assumes** *finite* (*A* :: *'a set*)
**and** *finite* (*B* :: *'a set*)
**shows** $\sum ((\lambda x.\ \sum (f\ x\ `\ B))\ `\ A) = \sum ((\lambda p.\ f\ (fst\ p)\ (snd\ p))\ `\ (A \times B))$
⟨*proof*⟩

**lemma** *flatten2-im*:
**fixes** *f* :: *'a* ⇒ *'a* ⇒ *'b*::*join-semilattice-zero*
**assumes** *finite* (*A* ::*'a set*)
**and** *finite* (*B* ::*'a set*)
**shows** $\sum ((\lambda y.\ \sum\ ((\lambda x.\ f\ x\ y)\ `\ A))\ `\ B) = \sum ((\lambda p.\ f\ (fst\ p)\ (snd\ p))\ `\ (A \times B))$

⟨*proof*⟩

**lemma** *sum-flatten1*:
**fixes** $f :: 'a \Rightarrow 'a \Rightarrow 'b$::*join-semilattice-zero*
**assumes** *finite* $(A :: 'a\ set)$
**and** *finite* $(B :: 'a\ set)$
**shows** $\sum \{\sum \{f\ x\ y\ |y.\ y \in B\}\ |x.\ x \in A\} = \sum \{f\ x\ y\ |x\ y.\ x \in A \wedge y \in B\}$
⟨*proof*⟩

**lemma** *sum-flatten2*:
**fixes** $f :: 'a \Rightarrow 'a \Rightarrow 'b$::*join-semilattice-zero*
**assumes** *finite* $A$
**and** *finite* $B$
**shows** $\sum \{\sum\ \{f\ x\ y\ |x.\ x \in A\}\ |y.\ y \in B\} = \sum \{f\ x\ y\ |x\ y.\ x \in A \wedge y \in B\}$
⟨*proof*⟩

Next we show another additivity property for suprema.

**lemma** *sum-fun-sum*:
**fixes** $f\ g :: 'a \Rightarrow 'b$::*join-semilattice-zero*
**assumes** *finite* $(A :: 'a\ set)$
**shows** $\sum ((\lambda x.\ f\ x\ +\ g\ x)\ `\ A) = \sum (f\ `\ A) + \sum (g\ `\ A)$
⟨*proof*⟩

The last lemma of this section prepares the distributivity laws that hold for dioids. It states that a strict additive function distributes over finite suprema, which is a continuity property in the finite.

**lemma** *sum-fun-add*:
**fixes** $f :: 'a$::*join-semilattice-zero* $\Rightarrow 'b$::*join-semilattice-zero*
**assumes** *finite* $(X :: 'a\ set)$
**and** *fstrict*: $f\ 0 = 0$
**and** *fadd*: $\bigwedge x\ y.\ f\ (x + y) = f\ x + f\ y$
**shows** $f\ (\sum\ X) = \sum (f\ `\ X)$
⟨*proof*⟩

## 14.3 Finite Suprema in Dioids

In this section we mainly prove variants of distributivity laws.

**lemma** *sum-distl*:
**assumes** *finite* $Y$
**shows** $(x :: 'a$::*dioid-one-zero*$) \cdot (\sum Y) = \sum \{x \cdot y | y.\ y \in Y\}$
⟨*proof*⟩

**lemma** *sum-distr*:
**assumes** *finite* $X$
**shows** $(\sum X) \cdot (y :: 'a$::*dioid-one-zero*$) = \sum \{x \cdot y | x.\ x \in X\}$
⟨*proof*⟩

**lemma** *sum-fun-distl*:

**fixes** $f :: 'a \Rightarrow 'b$::*dioid-one-zero*
**assumes** *finite* $(Y :: 'a\ set)$
**shows** $x \cdot \sum (f \mathbin{`} Y) = \sum \{x \cdot f\ y \mid y.\ y \in Y\}$
⟨*proof*⟩

**lemma** *sum-fun-distr*:
**fixes** $f :: 'a \Rightarrow 'b$::*dioid-one-zero*
**assumes** *finite* $(X :: 'a\ set)$
**shows** $\sum (f \mathbin{`} X) \cdot y = \sum \{f\ x \cdot y \mid x.\ x \in X\}$
⟨*proof*⟩

**lemma** *sum-distl-flat*:
**assumes** *finite* $(X :: 'a$::*dioid-one-zero set*$)$
**and** *finite* $Y$
**shows** $\sum \{x \cdot \sum Y \mid x.\ x \in X\} = \sum \{x \cdot y \mid x\ y.\ x \in X \wedge y \in Y\}$
⟨*proof*⟩

**lemma** *sum-distr-flat*:
**assumes** *finite* $X$
**and** *finite* $(Y :: 'a$::*dioid-one-zero set*$)$
**shows** $\sum \{(\sum X) \cdot y \mid y.\ y \in Y\} = \sum \{x \cdot y \mid x\ y.\ x \in X \wedge y \in Y\}$
⟨*proof*⟩

**lemma** *sum-sum-distl*:
**assumes** *finite* $(X :: 'a$::*dioid-one-zero set*$)$
**and** *finite* $Y$
**shows** $\sum ((\lambda x.\ x \cdot (\sum Y)) \mathbin{`} X) = \sum \{x \cdot y \mid x\ y.\ x \in X \wedge y \in Y\}$
⟨*proof*⟩

**lemma** *sum-sum-distr*:
**assumes** *finite* $X$
**and** *finite* $Y$
**shows** $\sum ((\lambda y.\ (\sum X) \cdot (y :: 'a$::*dioid-one-zero*$)) \mathbin{`} Y) = \sum \{x \cdot y \mid x\ y.\ x \in X \wedge y \in Y\}$
⟨*proof*⟩

**lemma** *sum-sum-distl-fun*:
**fixes** $f\ g :: 'a \Rightarrow 'b$::*dioid-one-zero*
**fixes** $h :: 'a \Rightarrow 'a\ set$
**assumes** $\bigwedge x.$ *finite* $(h\ x)$
**and** *finite* $X$
**shows** $\sum ((\lambda x.\ f\ x \cdot \sum (g \mathbin{`} h\ x)) \mathbin{`} X) = \sum \{\sum \{f\ x \cdot g\ y \mid y.\ y \in h\ x\} \mid x.\ x \in X\}$
⟨*proof*⟩

**lemma** *sum-sum-distr-fun*:
**fixes** $f\ g :: 'a \Rightarrow 'b$::*dioid-one-zero*
**fixes** $h :: 'a \Rightarrow 'a\ set$
**assumes** *finite* $Y$

**and** $\bigwedge y.$ *finite* $(h\ y)$
**shows** $\sum((\lambda y.\ \sum(f\ `\ h\ y)\cdot g\ y)\ `\ Y) = \sum\{\sum\{f\ x\cdot g\ y\ |x.\ x\in(h\ y)\}\ |y.\ y\in Y\}$
⟨*proof*⟩

**lemma** *sum-dist*:
**assumes** *finite* $(A :: 'a{::}$*dioid-one-zero set*$)$
**and** *finite* $B$
**shows** $(\sum A)\cdot(\sum B) = \sum\{x\cdot y\ |x\ y.\ x\in A\wedge y\in B\}$
⟨*proof*⟩

**lemma** *dioid-sum-prod-var*:
**fixes** $f\ g :: 'a\Rightarrow 'b{::}$*dioid-one-zero*
**assumes** *finite* $(A ::'a$ *set*$)$
**shows** $(\sum(f\ `\ A))\cdot(\sum\ (g\ `\ A)) = \sum\{f\ x\cdot g\ y\ |x\ y.\ x\in A\wedge y\in A\}$
⟨*proof*⟩

**lemma** *dioid-sum-prod*:
**fixes** $f\ g :: 'a\Rightarrow 'b{::}$*dioid-one-zero*
**assumes** *finite* $(A :: 'a$ *set*$)$
**shows** $(\sum\{f\ x\ |x.\ x\in A\})\cdot(\sum\{g\ x\ |x.\ x\in A\}) = \sum\{f\ x\cdot g\ y\ |x\ y.\ x\in A\wedge y\in A\}$
⟨*proof*⟩

**lemma** *sum-image*:
**fixes** $f :: 'a\Rightarrow 'b{::}$*join-semilattice-zero*
**assumes** *finite* $X$
**shows** *sum* $f\ X = \sum(f\ `\ X)$
⟨*proof*⟩

**lemma** *sum-interval-cong*:
$[\![\bigwedge i.\ [\![\ m\le i;\ i\le n\ ]\!]\Longrightarrow P(i) = Q(i)\ ]\!]\Longrightarrow(\sum i{=}m..n.\ P(i)) = (\sum i{=}m..n.\ Q(i))$
⟨*proof*⟩

**lemma** *sum-interval-distl*:
**fixes** $f ::$ *nat* $\Rightarrow 'a{::}$*dioid-one-zero*
**assumes** $m\le n$
**shows** $x\cdot(\sum i{=}m..n.\ f(i)) = (\sum i{=}m..n.\ (x\cdot f(i)))$
⟨*proof*⟩

**lemma** *sum-interval-distr*:
**fixes** $f ::$ *nat* $\Rightarrow 'a{::}$*dioid-one-zero*
**assumes** $m\le n$
**shows** $(\sum i{=}m..n.\ f(i))\cdot y = (\sum i{=}m..n.\ (f(i)\cdot y))$
⟨*proof*⟩

There are interesting theorems for finite sums in Kleene algebras; we leave them for future consideration.

**end**

# 15 Formal Power Series

**theory** *Formal-Power-Series*
**imports** *Finite-Suprema Kleene-Algebra*
**begin**

## 15.1 The Type of Formal Power Series

Formal powerseries are functions from a free monoid into a dioid. They have applications in formal language theory, e.g., weighted automata. As usual, we represent elements of a free monoid by lists.

This theory generalises Amine Chaieb's development of formal power series as functions from natural numbers, which may be found in *HOL/Library/Formal_Power_Series.thy*.

**typedef** $('a, 'b)$ *fps* $= \{f::'a\ list \Rightarrow 'b.\ True\}$
  **morphisms** *fps-nth Abs-fps*
  ⟨*proof*⟩

It is often convenient to reason about functions, and transfer results to formal power series.

**setup-lifting** *type-definition-fps*

**declare** *fps-nth-inverse* [*simp*]

**notation** *fps-nth* (**infixl** $ 75)

**lemma** *expand-fps-eq*: $p = q \longleftrightarrow (\forall n.\ p\ \$\ n = q\ \$\ n)$
⟨*proof*⟩

**lemma** *fps-ext*: $(\bigwedge n.\ p\ \$\ n = q\ \$\ n) \Longrightarrow p = q$
⟨*proof*⟩

**lemma** *fps-nth-Abs-fps* [*simp*]: *Abs-fps f* $ *n* = *f n*
⟨*proof*⟩

## 15.2 Definition of the Basic Elements 0 and 1 and the Basic Operations of Addition and Multiplication

The zero formal power series maps all elements of the monoid (all lists) to zero.

**instantiation** *fps* :: (*type*,*zero*) *zero*
**begin**
  **definition** *zero-fps* **where**
    *0* = *Abs-fps* $(\lambda n.\ 0)$
  **instance** ⟨*proof*⟩

**end**

**lemma** *fps-zero-nth* [*simp*]: *0 $ n = 0*
⟨*proof*⟩

The unit formal power series maps the monoidal unit (the empty list) to one and all other elements to zero.

**instantiation** *fps* :: (*type*,{*one*,*zero*}) *one*
**begin**
**definition** *one-fps* **where**
*1 = Abs-fps* (λ*n*. *if n = [] then 1 else 0*)
**instance** ⟨*proof*⟩
**end**

**lemma** *fps-one-nth-Nil* [*simp*]: *1 $ [] = 1*
⟨*proof*⟩

**lemma** *fps-one-nth-Cons* [*simp*]: *1 $ (x # xs) = 0*
⟨*proof*⟩

Addition of formal power series is the usual pointwise addition of functions.

**instantiation** *fps* :: (*type*,*plus*) *plus*
**begin**
**definition** *plus-fps* **where**
*f + g = Abs-fps* (λ*n*. *f $ n + g $ n*)
**instance** ⟨*proof*⟩
**end**

**lemma** *fps-add-nth* [*simp*]: *(f + g) $ n = f $ n + g $ n*
⟨*proof*⟩

This directly shows that formal power series form a semilattice with zero.

**lemma** *fps-add-assoc*: *((f*::(*'a*,*'b*::*semigroup-add*) *fps*) *+ g) + h = f + (g + h)*
⟨*proof*⟩

**lemma** *fps-add-comm* [*simp*]: *(f*::(*'a*,*'b*::*ab-semigroup-add*) *fps*) *+ g = g + f*
⟨*proof*⟩

**lemma** *fps-add-idem* [*simp*]: *(f*::(*'a*,*'b*::*join-semilattice*) *fps*) *+ f = f*
⟨*proof*⟩

**lemma** *fps-zerol* [*simp*]: *(f*::(*'a*,*'b*::*monoid-add*) *fps*) *+ 0 = f*
⟨*proof*⟩

**lemma** *fps-zeror* [*simp*]: *0 + (f*::(*'a*,*'b*::*monoid-add*) *fps*) *= f*
⟨*proof*⟩

The product of formal power series is convolution. The product of two formal powerseries at a list is obtained by splitting the list into all possible

prefix/suffix pairs, taking the product of the first series applied to the first coordinate and the second series applied to the second coordinate of each pair, and then adding the results.

**instantiation** *fps* :: (*type*,{*comm-monoid-add*,*times*}) *times*
**begin**
  **definition** *times-fps* **where**
    $f * g = Abs\text{-}fps\ (\lambda n.\ \sum \{f\ \$\ y * g\ \$\ z\ |y\ z.\ n = y\ @\ z\})$
  **instance** ⟨*proof*⟩
**end**

We call the set of all prefix/suffix splittings of a list *xs* the *splitset* of *xs*.

**definition** *splitset* **where**
  $splitset\ xs \equiv \{(p,\ q).\ xs = p\ @\ q\}$

Altenatively, splitsets can be defined recursively, which yields convenient simplification rules in Isabelle.

**fun** *splitset-fun* **where**
  $splitset\text{-}fun\ []\ \ = \{([],\ [])\}$
| $splitset\text{-}fun\ (x\ \#\ xs) = insert\ ([],\ x\ \#\ xs)\ (apfst\ (Cons\ x)\ `\ splitset\text{-}fun\ xs)$

**lemma** *splitset-consl*:
  $splitset\ (x\ \#\ xs) = insert\ ([],\ x\ \#\ xs)\ (apfst\ (Cons\ x)\ `\ splitset\ xs)$
⟨*proof*⟩

**lemma** *splitset-eq-splitset-fun*: $splitset\ xs = splitset\text{-}fun\ xs$
⟨*proof*⟩

The definition of multiplication is now more precise.

**lemma** *fps-mult-var*:
  $(f * g)\ \$\ n = \sum \{f\ \$\ (fst\ p) * g\ \$\ (snd\ p)\ |\ p.\ p \in splitset\ n\}$
⟨*proof*⟩

**lemma** *fps-mult-image*:
  $(f * g)\ \$\ n = \sum ((\lambda p.\ f\ \$\ (fst\ p) * g\ \$\ (snd\ p))\ `\ splitset\ n)$
⟨*proof*⟩

Next we show that splitsets are finite and non-empty.

**lemma** *splitset-fun-finite* [*simp*]: *finite* (*splitset-fun xs*)
  ⟨*proof*⟩

**lemma** *splitset-finite* [*simp*]: *finite* (*splitset xs*)
  ⟨*proof*⟩

**lemma** *split-append-finite* [*simp*]: *finite* $\{(p,\ q).\ xs = p\ @\ q\}$
  ⟨*proof*⟩

**lemma** *splitset-fun-nonempty* [*simp*]: $splitset\text{-}fun\ xs \neq \{\}$
  ⟨*proof*⟩

**lemma** *splitset-nonempty* [*simp*]: *splitset xs* $\neq$ {}
⟨*proof*⟩

We now proceed with proving algebraic properties of formal power series.

**lemma** *fps-annil* [*simp*]:
*0* $*$ (*f*::('*a*::*type*,'*b*::{*comm-monoid-add*,*mult-zero*}) *fps*) = *0*
⟨*proof*⟩

**lemma** *fps-annir* [*simp*]:
(*f*::('*a*::*type*,'*b*::{*comm-monoid-add*,*mult-zero*}) *fps*) $*$ *0* = *0*
⟨*proof*⟩

**lemma** *fps-distl*:
(*f*::('*a*::*type*,'*b*::{*join-semilattice-zero*,*semiring*}) *fps*) $*$ (*g* + *h*) = (*f* $*$ *g*) + (*f* $*$ *h*)
⟨*proof*⟩

**lemma** *fps-distr*:
((*f*::('*a*::*type*,'*b*::{*join-semilattice-zero*,*semiring*}) *fps*) + *g*) $*$ *h* = (*f* $*$ *h*) + (*g* $*$ *h*)
⟨*proof*⟩

The multiplicative unit laws are surprisingly tedious. For the proof of the left unit law we use the recursive definition, which we could as well have based on splitlists instead of splitsets.

However, a right unit law cannot simply be obtained along the lines of this proofs. The reason is that an alternative recursive definition that produces a unit with coordinates flipped would be needed. But this is difficult to obtain without snoc lists. We therefore prove the right unit law more directly by using properties of suprema.

**lemma** *fps-onel* [*simp*]:
*1* $*$ (*f*::('*a*::*type*,'*b*::{*join-semilattice-zero*,*monoid-mult*,*mult-zero*}) *fps*) = *f*
⟨*proof*⟩

**lemma** *fps-oner* [*simp*]:
(*f*::('*a*::*type*,'*b*::{*join-semilattice-zero*,*monoid-mult*,*mult-zero*}) *fps*) $*$ *1* = *f*
⟨*proof*⟩

Finally we prove associativity of convolution. This requires splitting lists into three parts and rearranging these parts in two different ways into splitsets. This rearrangement is captured by the following technical lemma.

**lemma** *splitset-rearrange*:
**fixes** *F* :: '*a list* ⇒ '*a list* ⇒ '*a list* ⇒ '*b*::*join-semilattice-zero*
**shows** $\sum \{\sum \{F\ (fst\ p)\ (fst\ q)\ (snd\ q) \mid q.\ q \in splitset\ (snd\ p)\} \mid p.\ p \in splitset\ x\} =$
$\sum \{\sum \{F\ (fst\ q)\ (snd\ q)\ (snd\ p) \mid q.\ q \in splitset\ (fst\ p)\} \mid p.\ p \in splitset\ x\}$

(**is** *?lhs* = *?rhs*)
⟨*proof*⟩

**lemma** *fps-mult-assoc*: (*f*::('*a*::*type*,'*b*::*dioid-one-zero*) *fps*) ∗ (*g* ∗ *h*) = (*f* ∗ *g*) ∗ *h*
⟨*proof*⟩

## 15.3 The Dioid Model of Formal Power Series

We can now show that formal power series with suitably defined operations form a dioid. Many of the underlying properties already hold in weaker settings, where the target algebra is a semilattice or semiring. We currently ignore this fact.

**subclass** (**in** *dioid-one-zero*) *mult-zero*
⟨*proof*⟩

**instantiation** *fps* :: (*type*,*dioid-one-zero*) *dioid-one-zero*
**begin**

**definition** *less-eq-fps* **where**
(*f*::('*a*,'*b*) *fps*) ≤ *g* ⟷ *f* + *g* = *g*

**definition** *less-fps* **where**
(*f*::('*a*,'*b*) *fps*) < *g* ⟷ *f* ≤ *g* ∧ *f* ≠ *g*

**instance**
⟨*proof*⟩

**end**

**lemma** *expand-fps-less-eq*: (*f*::('*a*,'*b*::*dioid-one-zero*) *fps*) ≤ *g* ⟷ (∀ *n*. *f* $ *n* ≤ *g* $ *n*)
⟨*proof*⟩

## 15.4 The Kleene Algebra Model of Formal Power Series

There are two approaches to define the Kleene star. The first one defines the star for a certain kind of (so-called proper) formal power series into a semiring or dioid. The second one, which is more interesting in the context of our algebraic hierarchy, shows that formal power series into a Kleene algebra form a Kleene algebra. We have only formalised the latter approach.

**lemma** *Sum-splitlist-nonempty*:
∑ {*f ys zs* |*ys zs*. *xs* = *ys* @ *zs*} = ((*f* [] *xs*)::'*a*::*join-semilattice-zero*) + ∑ {*f ys zs* |*ys zs*. *xs* = *ys* @ *zs* ∧ *ys* ≠ []}
⟨*proof*⟩

**lemma** (**in** *left-kleene-algebra*) *add-star-eq*:

$x + y \cdot y^\star \cdot x = y^\star \cdot x$
⟨*proof*⟩

**instantiation** *fps* :: (*type*,*kleene-algebra*) *kleene-algebra*
**begin**

We first define the star on functions, where we can use Isabelle's package for recursive functions, before lifting the definition to the type of formal power series.
This definition of the star is from an unpublished manuscript by Esik and Kuich.

**declare** *rev-conj-cong*[*fundef-cong*]
— required for the function package to prove termination of *star-fps-rep*

**fun** *star-fps-rep* **where**
*star-fps-rep-Nil*: *star-fps-rep f* [] = (*f* [])$^\star$
| *star-fps-rep-Cons*: *star-fps-rep f n* = (*f* [])$^\star$ · $\sum$ {*f y* · *star-fps-rep f z* |*y z*. *n* = *y* @ *z* ∧ *y* ≠ []}

**lift-definition** *star-fps* :: ('*a*, '*b*) *fps* ⇒ ('*a*, '*b*) *fps* **is** *star-fps-rep* ⟨*proof*⟩

**lemma** *star-fps-Nil* [*simp*]: *f*$^\star$ $ [] = (*f* $ [])$^\star$
⟨*proof*⟩

**lemma** *star-fps-Cons* [*simp*]: *f*$^\star$ $ (*x* # *xs*) = (*f* $ [])$^\star$ · $\sum$ {*f* $ *y* · *f*$^\star$ $ *z* |*y z*. *x* # *xs* = *y* @ *z* ∧ *y* ≠ []}
⟨*proof*⟩

**instance**
⟨*proof*⟩

**end**

**end**

# 16 Infinite Matrices

**theory** *Inf-Matrix*
**imports** *Finite-Suprema*
**begin**

Matrices are functions from two index sets into some suitable algebra. We consider arbitrary index sets, not necessarily the positive natural numbers up to some bounds; our coefficient algebra is a dioid. Our only restriction is that summation in the product of matrices is over a finite index set. This follows essentially Droste and Kuich's introductory article in the Handbook of Weighted Automata [10].

Under these assumptions we show that dioids are closed under matrix formation. Our proofs are similar to those for formal power series, but simpler.

**type-synonym** $('a, 'b, 'c)$ *matrix* $= 'a \Rightarrow 'b \Rightarrow 'c$

**definition** *mat-one* :: $('a, 'a, 'c::$*dioid-one-zero*$)$ *matrix* $(\varepsilon)$ **where**
$\varepsilon\ i\ j \equiv (if\ (i = j)\ then\ 1\ else\ 0)$

**definition** *mat-zero* :: $('a, 'b, 'c::$*dioid-one-zero*$)$ *matrix* $(\delta)$ **where**
$\delta \equiv \lambda j\ i.\ 0$

**definition** *mat-add* :: $('a, 'b, 'c::$*dioid-one-zero*$)$ *matrix* $\Rightarrow$ $('a, 'b, 'c)$ *matrix* $\Rightarrow$ $('a, 'b, 'c)$ *matrix* (**infixl** $\oplus$ *70*) **where**
$(f \oplus g) \equiv \lambda i\ j.\ (f\ i\ j) + (g\ i\ j)$

**lemma** *mat-add-assoc*: $(f \oplus g) \oplus h = f \oplus (g \oplus h)$
⟨*proof*⟩

**lemma** *mat-add-comm*: $f \oplus g = g \oplus f$
⟨*proof*⟩

**lemma** *mat-add-idem*[*simp*]: $f \oplus f = f$
⟨*proof*⟩

**lemma** *mat-zerol*[*simp*]: $f \oplus \delta = f$
⟨*proof*⟩

**lemma** *mat-zeror*[*simp*]: $\delta \oplus f = f$
⟨*proof*⟩

**definition** *mat-mult* :: $('a, 'k::$*finite*$, 'c::$*dioid-one-zero*$)$ *matrix* $\Rightarrow$ $('k, 'b, 'c)$ *matrix* $\Rightarrow$ $('a, 'b, 'c)$ *matrix* (**infixl** $\otimes$ *60*) **where**
$(f \otimes g)\ i\ j \equiv \sum \{(f\ i\ k) \cdot (g\ k\ j) \mid k.\ k \in UNIV\}$

**lemma** *mat-annil*[*simp*]: $\delta \otimes f = \delta$
⟨*proof*⟩

**lemma** *mat-annir*[*simp*]: $f \otimes \delta = \delta$
⟨*proof*⟩

**lemma** *mat-distl*: $f \otimes (g \oplus h) = (f \otimes g) \oplus (f \otimes h)$
⟨*proof*⟩

**lemma** *mat-distr*: $(f \oplus g) \otimes h = (f \otimes h) \oplus (g \otimes h)$
⟨*proof*⟩

**lemma** *logic-aux1*: $(\exists k.\ (i = k \longrightarrow x = f\ i\ j) \wedge (i \neq k \longrightarrow x = 0)) \longleftrightarrow (\exists k.\ i = k \wedge x = f\ i\ j) \vee (\exists k.\ i \neq k \wedge x = 0)$
⟨*proof*⟩

**lemma** *logic-aux2*: $(\exists\, k.\ (k = j \longrightarrow x = f\ i\ j) \wedge (k \neq j \longrightarrow x = 0)) \longleftrightarrow (\exists\, k.\ k = j \wedge x = f\ i\ j) \vee (\exists\, k.\ k \neq j \wedge x = 0)$
⟨*proof*⟩

**lemma** *mat-onel*[*simp*]: $\varepsilon \otimes f = f$
⟨*proof*⟩

**lemma** *mat-oner*[*simp*]: $f \otimes \varepsilon = f$
⟨*proof*⟩

**lemma** *mat-rearrange*:
**fixes** $F :: 'a \Rightarrow 'k1 \Rightarrow 'k2 \Rightarrow 'b \Rightarrow 'c$::*dioid-one-zero*
**assumes** *fUNk1*: *finite* $(UNIV::'k1\ set)$
**assumes** *fUNk2*: *finite* $(UNIV::'k2\ set)$
**shows** $\sum \{\sum \{F\ i\ k1\ k2\ j\ |k2.\ k2 \in (UNIV::'k2\ set)\}\ |k1.\ k1 \in (UNIV::'k1\ set)\} = \sum \{\sum \{F\ i\ k1\ k2\ j\ |k1.\ k1 \in UNIV\}\ |k2.\ k2 \in UNIV\}$
⟨*proof*⟩

**lemma** *mat-mult-assoc*: $f \otimes (g \otimes h) = (f \otimes g) \otimes h$
⟨*proof*⟩

**definition** *mat-less-eq* :: $('a, 'b, 'c$::*dioid-one-zero*$)$ *matrix* ⇒ $('a, 'b, 'c)$ *matrix* ⇒ *bool* **where**
*mat-less-eq f g* = $(f \oplus g = g)$

**definition** *mat-less* :: $('a, 'b, 'c$::*dioid-one-zero*$)$ *matrix* ⇒ $('a, 'b, 'c)$ *matrix* ⇒ *bool* **where**
*mat-less f g* = $(mat\text{-}less\text{-}eq\ f\ g \wedge f \neq g)$

**interpretation** *matrix-dioid*: *dioid-one-zero* *mat-add mat-mult mat-one mat-zero mat-less-eq mat-less*
⟨*proof*⟩

As in the case of formal power series we currently do not implement the Kleene star of matrices, since this is complicated.

**end**

# References


[1] A. Armstrong, S. Foster, W. Guttmann, G. Struth, and T. Weber. Relation algebra. *Archive of Formal Proofs*, 2014.

[2] A. Armstrong, V. B. F. Gomes, and G. Struth. Kleene algebra with tests. *Archive of Formal Proofs*, 2014.

[3] A. Armstrong and G. Struth. Automated reasoning in higher-order regular algebra. In W. Kahl and T. G. Griffin, editors, *RAMICS 2012*, vol-

ume 7560 of *Lecture Notes in Computer Science*, pages 66–81. Springer, 2012.

[4] L. Bachmair and N. Dershowitz. Commutation, transformation, and termination. In J. H. Siekmann, editor, *Conference on Automated Deduction*, volume 230 of *Lecture Notes in Computer Science*, pages 5–20. Springer, 1986.

[5] G. Birkhoff. *Lattice Theory*. American Mathematical Society Colloquium Publications, 1967.

[6] M. Boffa. Une remarque sur les systèmes complets d'identités rationnelles. *Informatique Theéorique et Applications*, 24(4):419–423, 1990.

[7] E. Cohen. Separation and reduction. In R. C. Backhouse and J. N. Oliveira, editors, *MPC*, volume 1837 of *Lecture Notes in Computer Science*, pages 45–59. Springer, 2000.

[8] J. H. Conway. *Regular Algebra and Finite Machines*. Chapman and Hall, 1971.

[9] J. Desharnais, B. Möller, and G. Struth. Kleene algebra with domain. *ACM Trans. Comput. Log.*, 7(4):798–833, 2006.

[10] M. Droste, W. Kuich, and H. Vogler, editors. *Handbook of Weighted Automata*. Springer, 2009.

[11] S. Foster and G. Struth. Automated analysis of regular algebra. In B. Gramlich, D. Miller, and U. Sattler, editors, *IJCAR 2012*, volume 7364 of *Lecture Notes in Computer Science*, pages 271–285. Springer, 2012.

[12] S. Foster and G. Struth. Regular algebras. *Archive of Formal Proofs*, 2014.

[13] S. Foster, G. Struth, and T. Weber. Automated engineering of relational and algebraic methods in Isabelle/HOL – (invited tutorial). In H. C. M. de Swart, editor, *RAMICS*, volume 6663 of *Lecture Notes in Computer Science*, pages 52–67. Springer, 2011.

[14] V. B. F. Gomes, W. Guttmann, P. Höfner, G. Struth, and T. Weber. Kleene algebra with domain. *Archive of Formal Proofs*, 2016.

[15] M. Gondran and M. Minoux. *Graphs, Dioids and Semirings: New Models and Algorithms*, volume 41 of *Operations Research/Computer Science Interfaces*. Springer, 2010.

[16] W. Guttmann, G. Struth, and T. Weber. Automating algebraic methods in Isabelle. In S. Qin and Z. Qiu, editors, *ICFEM 2011*, volume 6991 of *Lecture Notes in Computer Science*, pages 617–632. Springer, 2011.

[17] W. Guttmann, G. Struth, and T. Weber. A repository for Tarski-Kleene algebras. In P. Höfner, A. McIver, and G. Struth, editors, *ATE 2011*, volume 760 of *CEUR Workshop Proceedings*, pages 30–39. CEUR-WS.org, 2011.

[18] D. Haren, D. Kozen, and J. Tiuryn. *Dynamic Logic*. MIT Press, 2000.

[19] P. Höfner and G. Struth. Algebraic notions of nontermination: Omega and divergence in idempotent semirings. *J. Log. Algebr. Program.*, 79(8):794–811, 2010.

[20] D. Kozen. On Kleene algebras and closed semirings. In B. Rovan, editor, *Mathematical Foundations of Computer Science*, volume 452 of *Lecture Notes in Computer Science*, pages 26–47. Springer, 1990.

[21] D. Kozen. A completeness theorem for Kleene algebras and the algebra of regular events. *Inf. Comput.*, 110(2):366–390, 1994.

[22] D. Kozen. On Hoare logic and Kleene algebra with tests. *ACM Trans. Comput. Log.*, 1(1):60–76, 2000.

[23] A. Krauss and T. Nipkow. Proof pearl: Regular expression equivalence and relation algebra. *J. Autom. Reasoning*, 49(1):95–106, 2012.

[24] A. McIver and T. Weber. Towards automated proof support for probabilistic distributed systems. In G. Sutcliffe and A. Voronkov, editors, *LPAR*, volume 3835 of *Lecture Notes in Computer Science*, pages 534–548, 2005.

[25] G. Pilz. *Near-rings*. North-Holland, Amsterdam, second edition, 1983.

[26] K. Solin. Normal forms in total correctness for while programs and action systems. *J. Logic and Algebraic Programming*, 80(6):362–375, 2011.

[27] G. Struth. Abstract abstract reduction. *J. Log. Algebr. Program.*, 66(2):239–270, 2006.

[28] G. Struth. Left omega algebras and regular equations. *J. Log. Algebr. Program.*, 81(6):705–717, 2012.

[29] Y. Venema. Representation of game algebras. *Studia Logica*, 75(2):239–256, 2003.

[30] J. von Wright. From Kleene algebra to refinement algebra. In E. A. Boiten and B. Möller, editors, *MPC*, volume 2386 of *LNCS*, pages 233–262. Springer, 2002.

[31] J. von Wright. Towards a refinement algebra. *Science of Computer Programming*, 51(1–2):23–45, 2004.

[32] K. W. Wagner. Eine topologische Charakterisierung einiger Klassen regulärer Folgenmengen. *Elektronische Informationsverarbeitung und Kybernetik*, 13(9):473–487, 1977.