Kleene Algebra
Alasdair Armstrong, Georg Struth, Tjark Weber
September 19, 2015

Contents

1 Introductory Remarks 3

2 Signatures 3

3 Dioids 4

3.1 Join Semilattices . . . . . . . . . . . . . . . . . . . . . . . . . 4
3.2 Join Semilattices with an Additive Unit . . . . . . . . . . . . 6
3.3 Near Semirings . . . . . . . . . . . . . . . . . . . . . . . . . . 7
3.4 Variants of Dioids . . . . . . . . . . . . . . . . . . . . . . . . 7
3.5 Families of Nearsemirings with a Multiplicative Unit . . . . . 9
3.6 Families of Nearsemirings with Additive Units . . . . . . . . 9
3.7 Duality by Opposition . . . . . . . . . . . . . . . . . . . . . . 11
3.8 Selective Near Semirings . . . . . . . . . . . . . . . . . . . . . 11

4 Models of Dioids 12

4.1 The Powerset Dioid over a Monoid . . . . . . . . . . . . . . . 12
4.2 Language Dioids . . . . . . . . . . . . . . . . . . . . . . . . . 13
4.3 Relation Dioids . . . . . . . . . . . . . . . . . . . . . . . . . . 14
4.4 Trace Dioids . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14
4.5 Sets of Traces . . . . . . . . . . . . . . . . . . . . . . . . . . . 16
4.6 The Path Dioid . . . . . . . . . . . . . . . . . . . . . . . . . . 17
4.7 Path Models with the Empty Path . . . . . . . . . . . . . . . 17
4.8 Path Models without the Empty Path . . . . . . . . . . . . . 20
4.9 The Distributive Lattice Dioid . . . . . . . . . . . . . . . . . . 22
4.10 The Boolean Dioid . . . . . . . . . . . . . . . . . . . . . . . . 23
4.11 The Max-Plus Dioid . . . . . . . . . . . . . . . . . . . . . . . 23
4.12 The Min-Plus Dioid . . . . . . . . . . . . . . . . . . . . . . . 25

5 Matrices 28

5.1 Type Definition . . . . . . . . . . . . . . . . . . . . . . . . . . 28
5.2 0 and 1 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29
5.3 Matrix Addition . . . . . . . . . . . . . . . . . . . . . . . . . . 30
12 Finite Suprema

12.1 Auxiliary Lemmas ........................................... 89
12.2 Finite Suprema in Semilattices .............................. 89
12.3 Finite Suprema in Dioids ..................................... 93

13 Formal Power Series ........................................... 95

13.1 The Type of Formal Power Series ............................ 96
13.2 Definition of the Basic Elements 0 and 1 and the Basic Operations of Addition and Multiplication .................. 96
13.3 The Dioid Model of Formal Power Series .................... 101
13.4 The Kleene Algebra Model of Formal Power Series ........ 102

14 Infinite Matrices .................................................. 105

1 Introductory Remarks

These theory files 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 [12] or an overview article [16]. 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 and Hoare logics as well as infinitary and higher-order Kleene algebras [15, 1], and an alternative hierarchy of regular algebras and Kleene algebras [11]—orthogonal to the present one—have also been implemented. These are not covered by this repository.

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.

```isabelle
class star-op = 
  fixes star :: 'a ⇒ 'a (-· [101] 100)

class omega-op = 
  fixes omega :: 'a ⇒ 'a (-ω [101] 100)

class residual-r-op = 
  fixes residual-r :: 'a ⇒ 'a ⇒ 'a (infixr → 60)

class residual-l-op = 
  fixes residual-l :: 'a ⇒ 'a ⇒ 'a (infixl ← 60)
```

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.

```isabelle
class plus-ord = plus + ord + 
  assumes less-eq-def: x ≤ y ⇔ x + y = y 
  and less-def: x < y ⇔ x ≤ y ∧ x ≠ 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.

```haskell
class join_semilattice = plus_ord +
  assumes add_assoc' [simp]: (x + y) + z = x + (y + z)
  and add_comm [simp]: x + y = y + x
  and add_idem [simp]: x + x = x
begin

lemma add_left_comm [simp]:
  b + (a + c) = a + (b + c)
  unfolding add_assoc' [symmetric] by simp

lemma add_left_idem [simp]:
  a + (a + b) = a + b
  unfolding add_assoc' [symmetric] by simp

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
  fix x y z :: 'a
  show \(x < y \iff x \leq y \land \neg y \leq x\)
  by (metis add_comm less_def less_eq_def)
  show \(x \leq x\)
  by (metis add_idem less_eq_def)
  show \(x \leq y \implies y \leq z \implies x \leq z\)
  by (metis add_assoc' less_eq_def)
  show \(x \leq y \implies y \leq x \implies x = y\)
  by (metis add_comm less_eq_def)
qed

Next we show that joins are least upper bounds.

lemma add_ub1 [simp]: \(x \leq x + y\)
  by (metis add_assoc' add_idem less_eq_def)

lemma add_ub2 [simp]: \(y \leq x + y\)
  by (metis add_assoc' add_comm add_idem less_eq_def)

lemma add_lub_var: \(x \leq z \implies y \leq z \implies x + y \leq z\)
  by (metis add_assoc' less_eq_def)

lemma add_lub: \(x + y \leq z \iff x \leq z \land y \leq z\)
  by (metis add_lub_var add_ub1 add_ub2 order_trans)

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

lemma add_iso: \(x \leq y \implies x + z \leq y + z\)
  by (metis add_lub add_ub2 less_eq_def)
```
lemma add-iso-var: $x \leq y \rightarrow u \leq v \rightarrow x + u \leq y + v$
by (metis add-comm add-iso add-lub)

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: $x \leq y \iff (\exists z. x + z = y)$
proof
assume $x \leq y$

hence $x + y = y$
by (metis less-eq-def)

thus $\exists z. x + z = y$
by auto

next
assume $\exists z. x + z = y$
then obtain $c$ where $x + c = y$
by auto
also have $x + c \leq y$
by (metis calculation eq-refl)
thus $x \leq y$
by (metis add-ub1 calculation)
qed

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
apply unfold-locales
apply auto
apply (metis add-comm add-zero-l)
done

lemma zero-least [simp]: $0 \leq x$
by (metis add-zero-l less-eq-def)

lemma add-zero-r [simp]: $x + 0 = x$
by (metis add-comm add-zero-l)

lemma zero-unique [simp]: $x \leq 0 \iff x = 0$
by (metis zero-least eq-iff)
lemma no-trivial-inverse: $x \neq 0 \rightarrow \neg(\exists y. x + y = 0)$
  by (metis zero-unique order-prop)

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 [23] 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': $(x + y) \cdot z = x \cdot z + y \cdot z$

subclass (in semiring) ab-near-semiring
  by (unfold-locales, metis distrib-right')

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 [3], but deviate from the definitions in Gondran and Minoux’s book [14].

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
by unfold-locales (auto simp add: add.commute add.left-commute)

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

lemma mult-isor: $x \leq y \rightarrow x \cdot z \leq y \cdot z$
proof
  assume $x \leq y$
hence \( x + y = y \)
by \( \text{metis \ less-eq-def} \)
also have \( x \cdot z + y \cdot z = (x + y) \cdot z \)
by \( \text{metis distrib-right} \)
moreover have \( ... = y \cdot z \)
by \( \text{metis calculation} \)
thus \( x \cdot z \leq y \cdot z \)
by \( \text{metis calculation less-eq-def} \)
qed

lemma \( x \leq y \longrightarrow z \cdot x \leq z \cdot y \)

nitpick \[\text{expect=genuine}\] — 3-element counterexample

oops

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

lemma \( \text{mult-isol-equiv-subdistl} \):
\[(\forall x \ y \ z. \ x \leq y \longrightarrow z \cdot x \leq z \cdot y) \leftrightarrow (\forall x \ y \ z. \ z \cdot x \leq z \cdot (x + y))\]
by \( \text{metis add-ub1 \ less-eq-def} \)

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 \[22\] and game algebras \[27\]. We are not aware that these structures have a special name, so we baptise them \( \text{pre-dioids} \).

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

class \( \text{pre-dioid = near-dioid +} \)
assumes \( \text{subdistl: } z \cdot x \leq z \cdot (x + y) \)
begin

Now, obviously, left isotonicity follows from left subdistributivity.

lemma \( \text{subdistl-var: } z \cdot x + z \cdot y \leq z \cdot (x + y) \)
by \( \text{metis add.commute \ add-lub \ subdistl} \)

lemma \( \text{mult-isol: } x \leq y \longrightarrow z \cdot x \leq z \cdot y \)

proof
assume \( x \leq y \)
hence \( x + y = y \)
by \( \text{metis \ less-eq-def} \)
also have \( z \cdot x + z \cdot y \leq z \cdot (x + y) \)
by \( \text{metis \ subdistl-var} \)
moreover have \( ... = z \cdot y \)
by \( \text{metis \ calculation} \)
thus \( z \cdot x \leq z \cdot y \)
by \( \text{metis \ add-ub1 \ calculation \ order-trans} \)

proof

qed

**lemma** mult-isol-var: $u \leq x \land v \leq y \longrightarrow u \cdot v \leq x \cdot y$
by (metis mult-isol mult-isor order-trans)

**lemma** mult-double-iso: $x \leq y \longrightarrow w \cdot x \cdot z \leq w \cdot y \cdot z$
by (metis mult-isol mult-isor)

**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
by (unfold-locales, metis order-prop distrib-left)

### 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
by (unfold-locales, simp-all)

**end**

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

For near dioids with one, it would be sufficient to require $1 + 1 = 1$. This implies $x + x = x$ for arbitrary $x$ (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 ..
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 ≠ 1.

lemma add-zeror [simp]; x + 0 = x
  by (metis add.commute add-zerol)

end
```

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

subclass (in near-dioid-one-zerol) join-semilattice-zero
  by (unfold-locales, metis add-zerol)

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

subclass (in pre-dioid-one-zerol) near-dioid-one-zerol ..

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 ..

We now make zero also a right annihilator.

class ab-near-semiring-one-zero = ab-near-semiring-one-zerol +
assumes annir [simp]; x · 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 ..
class \textit{diodoid-one-zero} = \textit{diodoid-one-zero} + \textit{ab-near-semiring-one-zero} \\
subclass \textbf{(in \textit{diodoid-one-zero})} \textit{pre-diodoid-one-zero} .. \\
subclass \textbf{(in \textit{diodoid-one-zero})} \textit{semiring-one-zero} .. \\

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 \textbf{(in \textit{times})} \textit{opp-mult (infixl \odot 70)} \where \textit{x \odot y \equiv y \cdot x} \\
lemma \textbf{(in \textit{semiring-1})} \textit{dual-semiring-1}:
\textit{class \textit{semiring-1} 1 \textit{op} \odot \textit{op} + 0}
\by \text{unfold-locales (auto simp add: opp-mult-def mult.assoc distrib-right distrib-left)}

lemma \textbf{(in \textit{diodoid-one-zero})} \textit{dual-diodoid-one-zero}:
\textit{class \textit{diodoid-one-zero} \textit{op} + \textit{op} \odot \textit{op} \leq \textit{op} <}
\by \text{unfold-locales (auto simp add: opp-mult-def mult.assoc distrib-right distrib-left)}

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 \textit{selective-near-semiring} = \textit{ab-near-semiring} + \textit{plus-ord} + 
\assumes \textit{select}: \textit{x} + \textit{y} = \textit{x} \lor \textit{x} + \textit{y} = \textit{y}
\begin 
lemma \textit{select-alt}: \textit{x} + \textit{y} \in \{\textit{x}, \textit{y}\} 
\by \text{(metis insert-iff select)} 

\text{It follows immediately that every selective near semiring is a near dioid.}

subclass \textit{near-diodoid} 
\by \text{(unfold-locales, metis select)}

\text{Moreover, the order in a selective near semiring is obviously linear.}

subclass \textit{linorder} 
\by \text{(unfold-locales, metis add.commute add-ub1 select)}
\end 

class \textit{selective-semiring} = \textit{selective-near-semiring} + \textit{semiring-one-zero} 
\begin 
subclass \textit{diodoid-one-zero} ..
4 Models of Dioids

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 [8].

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.

```
theory Doid-Models
imports Doid 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 [8].

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) dioid-one-zero

begin

definition zero-set-def:
\[ \emptyset = \{\} \]

definition plus-set-def:
\[ A + B = A \cup B \]

instance

proof
fix \( X \ Y \ Z :: 'a \text{ set} \)

show \( X + Y + Z = X + (Y + Z) \)
by (simp add: Un-assoc plus-set-def)

show \( X + Y = Y + X \)
by (simp add: Un-commute plus-set-def)

show \( (X + Y) \cdot Z = X \cdot Z + Y \cdot Z \)
by (auto simp add: plus-set-def c-prod-def)

show \( 1 \cdot X = X \)
by (simp add: one-set-def c-prod-def)

show \( X \cdot 1 = X \)
by (simp add: one-set-def c-prod-def)

show \( \emptyset + X = X \)
by (simp add: plus-set-def zero-set-def)

show \( \emptyset \cdot X = \emptyset \)
by (simp add: c-prod-def zero-set-def)

show \( X \cdot \emptyset = \emptyset \)
by (simp add: c-prod-def zero-set-def)

show \( X \subseteq Y \iff X + Y = Y \)
by (simp add: plus-set-def subset-Un-eq)

show \( X \subset Y \iff X \subseteq Y \land X \neq Y \)
by (fact psubset-eq)

show \( X + X = X \)
by (simp add: Un-absorb plus-set-def)

show \( X \cdot (Y + Z) = X \cdot Y + X \cdot Z \)
by (auto simp add: plus-set-def c-prod-def)

qed

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 $\otimes$ denotes word concatenation.

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

begin

definition times-list-def:
$xs \ast ys \equiv xs \otimes ys$

definition one-list-def:
$1 \equiv []$

instance proof

fix $xs ys zs :: 'a list$
show $xs \ast ys \ast zs = xs \ast (ys \ast zs)$
by (simp add: times-list-def)

show $1 \ast xs = xs$
by (simp add: one-list-def times-list-def)

show $xs \ast 1 = xs$
by (simp add: one-list-def times-list-def)

qed

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 $\subseteq$ op $\subset$..  

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 $\subset$..  
by (unfold-locales, auto)

**interpretation** rel-monoid: monoid-mult Id op $O$ ..

4.4 Trace Dioids

Traces have been considered, for instance, by Kozen [20] 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 × ('a × 'p) list

**definition** first :: ('p, 'a) trace ⇒ 'p where
  first = fst

**lemma** first-conv [simp]: first (p, xs) = p
  by (unfold first-def, simp)

**fun** last :: ('p, 'a) trace ⇒ 'p where
  last (p, []) = p
  | last (-, xs) = snd (List.last xs)

**lemma** last-append [simp]: last (p, xs @ ys) = last (last (p, xs), ys)
  proof (cases xs)
  show xs = [] ⇒ last (p, xs @ ys) = last (last (p, xs), ys)
    by simp
  show ∀a list. xs = a # list ⇒
    last (p, xs @ ys) = last (last (p, xs), ys)
    proof (cases ys)
    show ∀a list. xs = a # list; ys = []
      ⇒ last (p, xs @ ys) = last (last (p, xs), ys)
      by simp
    show ∀a list aa lista. [xs = a # list; ys = aa # lista]
      ⇒ last (p, xs @ ys) = last (last (p, xs), ys)
      by simp
  qed
  qed

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 ⇒ ('p, 'a) trace ⇒ ('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
  by (cases x, simp add: t-fusion-def)
lemma fusion-rightneutral [simp]: t-fusion x (last x, []) = x
  by (simp add: t-fusion-def)

lemma first-t-fusion [simp]: last x = first y =⇒ first (t-fusion x y) = first x
  by (simp add: first-def t-fusion-def)

lemma last-t-fusion [simp]: last x = first y =⇒ last (t-fusion x y) = last y
  by (metis (lifting) Dioid-Models.last-append first-def t-fusion-def pair-collapse)

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
  by (cases x, cases y, cases z, simp add: t-fusion-def)

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)
  by (unfold t-prod-def) auto

lemma t-prod-intro [simp, intro]:
  [ u ∈ X; v ∈ Y; last u = first v ] =⇒ t-fusion u v ∈ X·Y
  by (metis (lifting) Dioid-Models.last-append first-def t-fusion-def pair-collapse)

lemma t-prod-elim [elim]:
  w ∈ X·Y =⇒ ∃ u v. w = t-fusion u v ∧ u ∈ X ∧ v ∈ Y ∧ last u = first v
  by (metis (lifting) Dioid-Models.last-append first-def t-fusion-def pair-collapse)
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** \(\text{trace-dioid}: \text{dioid-one-zero} \cup \text{t-prod} \subseteq \text{op} \subset\)

**apply** unfold-locale

**apply** (auto simp add: \(\text{t-prod-def} \ \text{t-one-def} \ \text{t-zero-def} \ \text{t-fusion-def}\))

**apply** (metis last-append)

**apply** (metis last-append append-assoc)

**done**

**no-notation**

\(\text{t-prod} \ (\text{infixl} \cdot 70)\)

### 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** \(\text{p-fusion} :: `\text{a path} \Rightarrow `\text{a path} \Rightarrow `\text{a path}\) where

\[
\begin{align*}
\text{p-fusion} \quad [] & = [] \\
\text{p-fusion} \quad [p] & = [] \\
\text{p-fusion} \quad [ps \ (q \# qs)] & = ps @ qs
\end{align*}
\]

**lemma** \(\text{p-fusion-assoc}: \text{p-fusion} \quad \text{ps} \ (\text{p-fusion} \quad \text{qs} \ \text{rs}) = \text{p-fusion} \ (\text{p-fusion} \quad \text{ps} \ \text{qs}) \ \text{rs}\)

**proof** (induct \(\text{rs}\))

**case** Nil show ?case

\(\text{by (metis list.exhaust \text{p-fusion.simps}1 \text{p-fusion.simps}2)}\)

**case** Cons show ?case

**proof** (induct \(\text{qs}\))

**case** Nil show ?case

\(\text{by (metis neg-Nil-conv \text{p-fusion.simps}1 \text{p-fusion.simps}2)}\)
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
  by (metis (hide-lams, no-types) List.last.simps List.last-append append-Nil2 assms)

**lemma** p-fusion-hd: 
<table>
<thead>
<tr>
<th>ps ≠ []; qs ≠ []</th>
</tr>
</thead>
<tbody>
<tr>
<td>⇒ hd (p-fusion ps qs) = hd ps</td>
</tr>
</tbody>
</table>
  by (metis list.exhaust p-fusion.simps(1) append-Cons list.sel(1))

**lemma** nonempty-p-fusion: 
<table>
<thead>
<tr>
<th>ps ≠ []; qs ≠ []</th>
</tr>
</thead>
<tbody>
<tr>
<td>⇒ p-fusion ps qs ≠ []</td>
</tr>
</tbody>
</table>
  by (metis list.exhaust append-Cons p-fusion.simps(3) list.simps(2))

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) |
  by (unfold p-prod-def) auto

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** (rule set-eqI)
  fix ps
  show ps ∈ (X · Y) · Z ←→ ps ∈ X · (Y · Z)
  **proof** (cases ps)
  case Nil thus ?thesis
    by auto (metis nonempty-p-fusion p-prod-iff)
  next
  case Cons thus ?thesis
by (auto simp add: p-prod-iff) (metis (hide-lams, mono-tags) nonempty-p-fusion
p-fusion-assoc p-fusion-hd p-fusion-last)+
qed
qed

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 (rule set-eqI)
  fix ps
  show ps ∈ p-one · X ←→ ps ∈ X
  proof (cases ps)
    case Nil thus ?thesis
    by (auto simp add: p-one-def p-prod-def, metis nonempty-p-fusion not-Cons-self)
  next
    case Cons thus ?thesis
    by (auto simp add: p-one-def p-prod-def, metis append-Cons append-Nil
list.sel(1) neq-Nil-conv p-fusion.simps(3), metis Cons-eq-appendI list.sel(1) last-ConsL
list.simps(3) p-fusion.simps(3) self-append-conv2)
  qed

lemma p-prod-oner [simp]: X · p-one = X
proof (rule set-eqI)
  fix ps
  show ps ∈ X · p-one ←→ ps ∈ X
  proof (cases ps)
    case Nil thus ?thesis
    by (auto simp add: p-one-def p-prod-def, metis nonempty-p-fusion not-Cons-self2,
metis p-fusion.simps(1))
  next
    case Cons thus ?thesis
    by (auto simp add: p-one-def p-prod-def, metis append-Nil2 neq-Nil-conv
p-fusion.simps(3), metis list.sel(1) list.simps(2) p-fusion.simps(3) self-append-conv)
  qed

Next we show distributivity laws at the powerset level.
lemma p-prod-distl: X · (Y ∪ Z) = X · Y ∪ X · Z
proof (rule set-eqI)
  fix ps
  show ps ∈ X · (Y ∪ Z) ←→ ps ∈ X · Y ∪ X · Z
  by (cases ps) (auto simp add: p-prod-iff)
qed
lemma p-prod-distr: \((X \cup Y) \cdot Z = X \cdot Z \cup Y \cdot Z\)

proof (rule set-eqI)
  fix ps
  show \(ps \in (X \cup Y) \cdot Z \iff ps \in X \cdot Z \cup Y \cdot Z\)
  by (cases ps) (auto simp add: p-prod-iff)
qed

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
  fix x y z :: 'a path set
  show \(x \cup y \cup z = x \cup (y \cup z)\)
  by auto
  show \((x \cdot y) \cdot z = x \cdot (y \cdot z)\)
  by (fact p-prod-assoc)
  show \((x \cup y) \cdot z = x \cdot z \cup y \cdot z\)
  by (fact p-prod-distr)
  show \(p\cdotone \cdot x = x\)
  by (fact p-prod-onel)
  show \(x \cdot p\cdotone = x\)
  by (fact p-prod-oner)
  show \(\{}\cup x = x\)
  by auto
  show \(\{} \cdot x = \{}\)
  by (metis all-not-in-conv p-prod-iff)
  show \(x \cdot \{} = \{}\)
  by (metis all-not-in-conv p-prod-iff)
  show \((x \subseteq y) = (x \cup y = y)\)
  by auto
  show \((x \subset y) = (x \subseteq y \land x \neq y)\)
  by auto
  show \(x \cup x = x\)
  by auto
  show \(x \cdot (y \cup z) = x \cdot y \cup x \cdot z\)
  by (metis p-prod-distl)
qed

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 \textit{pp-first} :: 'a ppath \Rightarrow 'a where
\begin{align*}
\text{pp-first} \ (\text{Node} \ x) & = x \\
| \text{pp-first} \ (\text{Cons} \ x \ -) & = x
\end{align*}

primrec \textit{pp-last} :: 'a ppath \Rightarrow 'a where
\begin{align*}
\text{pp-last} \ (\text{Node} \ x) & = x \\
| \text{pp-last} \ (\text{Cons} \ - \ xs) & = \text{pp-last} \ xs
\end{align*}

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

primrec \textit{pp-fusion} :: 'a ppath \Rightarrow 'a ppath \Rightarrow 'a ppath where
\begin{align*}
\text{pp-fusion} \ (\text{Node} \ x) \ ys & = ys \\
| \text{pp-fusion} \ (\text{Cons} \ x \ xs) \ ys & = \text{Cons} \ x \ (\text{pp-fusion} \ xs \ ys)
\end{align*}

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.

\textbf{lemma} \textit{pp-fusion-leftneutral} [simp]: \text{pp-fusion} \ (\text{Node} \ (\text{pp-first} \ x)) \ x = x
\textbf{by simp}

\textbf{lemma} \textit{pp-fusion-rightneutral} [simp]: \text{pp-fusion} \ x \ (\text{Node} \ (\text{pp-last} \ x)) = x
\textbf{by (induct x) simp-all}

\textbf{lemma} \textit{pp-first-pp-fusion} [simp]:
\begin{align*}
\text{pp-last} \ x = \text{pp-first} \ y \implies \text{pp-first} \ (\text{pp-fusion} \ x \ y) & = \text{pp-first} \ x \\
\textbf{by (induct x) simp-all}
\end{align*}

\textbf{lemma} \textit{pp-last-pp-fusion} [simp]:
\begin{align*}
\text{pp-last} \ x = \text{pp-first} \ y \implies \text{pp-last} \ (\text{pp-fusion} \ x \ y) & = \text{pp-last} \ y \\
\textbf{by (induct x) simp-all}
\end{align*}

\textbf{lemma} \textit{pp-fusion-assoc} [simp]:
\begin{align*}
\left[ \begin{array}{c}
\text{pp-last} \ x = \text{pp-first} \ y \\
\text{pp-last} \ y = \text{pp-first} \ z
\end{array} \right] \implies \text{pp-fusion} \ x \ (\text{pp-fusion} \ y \ z) & = \text{pp-fusion} \ (\text{pp-fusion} \ x \ y) \ z \\
\textbf{by (induct x) simp-all}
\end{align*}

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

\textbf{definition} \textit{pp-prod} :: 'a ppath set \Rightarrow 'a ppath set \Rightarrow 'a ppath set (infixl \cdot 70)
\textbf{where}
\begin{align*}
X \cdot Y = \{ \text{pp-fusion} \ u \ v \mid u \ v. \ u \in X \land v \in Y \land \text{pp-last} \ u = \text{pp-first} \ v \}
\end{align*}

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

\textbf{definition} \textit{pp-one} :: 'a ppath set
\textbf{where}
\begin{align*}
\text{pp-one} & \equiv \text{range} \ \text{Node}
\end{align*}
We again provide an elimination rule.

**Lemma pp-prod-if:**

\[ w \in X \cdot Y \iff (\exists u \, v. \ w = \text{pp-fusion} \ u \ , \ v \in X \ \land \ v \in Y \ \land \ \text{pp-last} \ u = \text{pp-first} \ v) \]

by (unfold pp-prod-def) auto

**Interpretation pp-path-dioid:**

\[ \text{dioid-one-zero} \cup \text{op} \cdot \text{pp-one} \{\} \ \text{op} \subseteq \text{op} \subset \]

**Proof**

fix \( x \, y \, z :: \text{`a ppath set} \)

show \( x \cup y \cup z = x \cup (y \cup z) \)

by auto

show \( x \cdot y \cdot z = x \cdot (y \cdot z) \)


show \( (x \cup y) \cdot z = x \cdot z \cup y \cdot z \)

by (auto simp add: pp-prod-def)

show \( \text{pp-one} \cdot x = x \)


show \( x \cdot \text{pp-one} = x \)


show \( \{\} \cup x = x \)

by auto

show \( \{\} \cdot x = \{\} \)

by (simp add: pp-prod-def)

show \( x \cdot \{\} = \{\} \)

by (simp add: pp-prod-def)

show \( x \subseteq y \iff x \cup y = y \)

by auto

show \( x \subseteq y \iff x \subseteq y \land x \neq y \)

by auto

show \( x \cup x = x \)

by auto

show \( x \cdot (y \cup z) = x \cdot y \cup x \cdot z \)

by (auto simp add: pp-prod-def)

qed

**No-Notation**

`pp-prod (infixl \cdot 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.

\textbf{class} bounded-distributive-lattice = bounded-lattice + distrib-lattice

\textbf{sublocale} bounded-distributive-lattice ⊆ dioid-one-zero sup inf top bot less-eq

\textbf{proof}

fix \ x \ y \ z

show \sup (\sup \ x \ y) \ z = \sup \ x \ (\sup \ y \ z)

by (fact sup-assoc)

show \sup \ x \ y = \sup \ y \ x

by (fact sup.commute)

show \inf (\inf \ x \ y) \ z = \inf \ x \ (\inf \ y \ z)

by (metis inf.commute inf.left-commute)

show \inf \ (\sup \ x \ y) \ z = \sup \ (\inf \ x \ z) \ (\inf \ y \ z)

by (fact inf-sup-distrib2)

show \inf \ top \ x = x

by simp

show \inf \ x \ top = x

by simp

show \sup \ bot \ x = x

by simp

show \inf \ bot \ x = bot

by simp

show \inf \ x \ bot = bot

by simp

show \(x \leq y\) = \(\sup \ x \ y = y\)

by (fact le-iff-sup)

show \(x < y\) = \((x \leq y \land x \neq y)\)

by auto

show \sup \ x \ x = x

by simp

show \inf \ x \ (\sup \ y \ z) = \sup \ (\inf \ x \ y) \ (\inf \ x \ z)

by (fact inf-sup-distrib1)

qed

4.10 The Boolean Dioid

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

\textbf{instantiation} bool :: bounded-distributive-lattice

\textbf{begin}

instance ..

\textbf{end}

\textbf{interpretation} boolean-dioid: dioid-one-zero sup inf True False less-eq less

by (unfold-locales, simp-all add: inf-bool-def sup-bool-def)
4.11 The Max-Plus Dioid

The following dioids have important applications in combinatorial optimisation, 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
  by (cases y, simp-all)

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 ≡ mreal-max x y

definition times-mreal-def:
  x * y ≡ mreal-plus x y

definition less-eq-mreal-def:
  (x::mreal) ≤ y ≡ x + y = y

definition less-mreal-def:
  (x::mreal) < y ≡ x ≤ y ∧ x ≠ y

24
instance
proof
fix x y z :: mreal
show x + y + z = x + (y + z)
  by (cases x, cases y, cases z, simp-all add: plus-mreal-def)
show x + y = y + x
  by (cases x, cases y, simp-all add: plus-mreal-def)
show x * y * z = x * (y * z)
  by (cases x, cases y, cases z, simp-all add: times-mreal-def)
show (x + y) * z = x * z + y * z
  by (cases x, cases y, cases z, simp-all add: plus-mreal-def times-mreal-def)
show 1 * x = x
  by (cases x, simp-all add: one-mreal-def times-mreal-def)
show x * 1 = x
  by (cases x, simp-all add: one-mreal-def times-mreal-def)
show 0 + x = x
  by (cases x, simp-all add: plus-mreal-def zero-mreal-def)
show 0 * x = 0
  by (cases x, simp-all add: times-mreal-def zero-mreal-def)
show x * 0 = 0
  by (cases x, simp-all add: times-mreal-def zero-mreal-def)
show x ≤ y ↔ x + y = y
  by (metis less-eq-mreal-def)
show x < y ↔ x ≤ y ∧ x ≠ y
  by (metis less-mreal-def)
show x + y = x ∨ x + y = y
  by (cases x, cases y, simp-all add: plus-mreal-def, metis linorder-le-cases
max.absorb-iff2 max.absorb1)
  show x * (y + z) = x * y + x * z
    by (cases x, cases y, cases z, simp-all add: plus-mreal-def times-mreal-def)
qed

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
  by (cases y, simp-all)
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 ≡ preal-min x y

  definition times-preal-def:
    x * y ≡ preal-plus x y

  definition less-eq-preal-def:
    (x::preal) ≤ y ≡ x + y = y

  definition less-preal-def:
    (x::preal) < y ≡ x ≤ y ∧ x ≠ y

instance
proof
  fix x y z :: preal
  show x + y + z = x + (y + z)
    by (cases x, cases y, cases z, simp-all add: plus-preal-def)
  show x + y = y + x
    by (cases x, cases y, simp-all add: plus-preal-def)
  show x * y * z = x * (y * z)
    by (cases x, cases y, cases z, simp-all add: times-preal-def)
  show (x + y) * z = x * z + y * z
    by (cases x, cases y, cases z, simp-all add: times-preal-def)
  show 1 * x = x
    by (cases x, simp-all add: one-preal-def times-preal-def)
  show x * 1 = x
    by (cases x, simp-all add: one-preal-def times-preal-def)
  show 0 + x = x
    by (cases x, simp-all add: plus-preal-def zero-preal-def)
  show 0 * x = 0
    by (cases x, simp-all add: times-preal-def zero-preal-def)
  show x * 0 = 0
    by (cases x, simp-all add: times-preal-def zero-preal-def)
  show x ≤ y ←→ x + y = y
    by (metis less-eq-preal-def)
  show x < y ←→ x ≤ y ∧ x ≠ y
by (metis less-preal-def)

show \( x + y = x \lor x + y = y \)
  by (cases x, cases y, simp-all add: plus-preal-def, metis linorder-le-cases
  min.absorb2 min.absorb-iff1)

show \( x \cdot (y + z) = x \cdot y + x \cdot z \)
  by (cases x, cases y, cases z, simp-all add: plus-preal-def times-preal-def)

qed

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
  by (cases y, simp-all)

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 \equiv PInfty

definition one-pnat-def:
  1 \equiv pnat 0

definition plus-pnat-def:
  \( x + y \equiv pnat-min x y \)

definition times-pnat-def:
  \( x \cdot y \equiv pnat-plus x y \)

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

definition less-pnat-def:
  \((x::pnat) < y \equiv x \leq y \land x \neq y\)

lemma zero-pnat-top: (x::pnat) \leq 1
  by (cases x, simp-all add: less-eq-pnat-def plus-pnat-def one-pnat-def)
instance
proof
  fix x y z :: pnat
  show x + y + z = x + (y + z)
    by (cases x, cases y, cases z, simp-all add: plus-pnat-def)
  show x + y = y + x
    by (cases x, cases y, simp-all add: plus-pnat-def)
  show x * y * z = x * (y * z)
    by (cases x, cases y, cases z, simp-all add: times-pnat-def)
  show (x + y) * z = x * z + y * z
    by (cases x, cases y, cases z, simp-all add: plus-pnat-def times-pnat-def)
  show 1 * x = x
    by (cases x, simp-all add: one-pnat-def times-pnat-def)
  show x * 1 = x
    by (cases x, simp-all add: one-pnat-def times-pnat-def)
  show 0 + x = x
    by (cases x, simp-all add: plus-pnat-def zero-pnat-def)
  show 0 * x = 0
    by (cases x, simp-all add: plus-pnat-def zero-pnat-def)
  show x * 0 = 0
    by (cases x, simp-all add: times-pnat-def zero-pnat-def)
  show x ≤ y ⟷ x + y = y
    by (metis less-eq-pnat-def)
  show x < y ⟷ x ≤ y ∧ x ≠ y
    by (metis less-pnat-def)
  show x + y = x ∨ x + y = y
    by (cases x, cases y, simp-all add: plus-pnat-def, metis linorder-le-cases min.absorb2 min.absorb-iff1)
  show x * (y + z) = x * y + x * z
    by (cases x, cases y, cases z, simp-all add: plus-pnat-def times-pnat-def)
qed

end
end

5 Matrices

theory Matrix
imports ~/src/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 [6].


5.1 Type Definition

typedef 'a atMost = {..<len-of TYPE('a::len)}
by auto

declare Rep-atMost-inject [simp]

lemma UNIV-atMost: (UNIV::{'a atMost set}) = Abs-atMost {..<len-of TYPE('a::len)}
apply auto
apply (rule Abs-atMost-induct)
apply auto
done

lemma finite-UNIV-atMost [simp]: finite (UNIV::{'a::len) atMost set)
by (simp add: UNIV-atMost)

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.

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

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

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 (λi j. 0)
instance ..
end

instantiation sqmatrix :: (zero,type) zero
begin
definition zero-sqmatrix-def: 0 ≡ SqMatrix (λi j. 0)
instance ..
end

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

instantiation matrix :: ({zero,one},len,len) one
begin
definition one-matrix-def:
\[ 1 \equiv \text{Matrix} \left( \lambda_{i,j} \text{ if } \text{Rep-atMost } i = \text{Rep-atMost } j \text{ then } 1 \text{ else } 0 \right) \]

\text{instance} ..
\text{end}

\text{instantiation} \text{ sqmatrix} :: (\{\text{zero, one}\}, \text{type}) \text{ one}
\text{begin}
\text{definition} \text{ one-sqmatrix-def}:
\quad 1 \equiv \text{SqMatrix} \left( \lambda_{i,j} \text{ if } i = j \text{ then } 1 \text{ else } 0 \right)
\text{instance} ..
\text{end}

### 5.3 Matrix Addition

\text{fun} \text{ matrix-plus where}
\text{matrix-plus} (\text{Matrix} A) (\text{Matrix} B) = \text{Matrix} \left( \lambda_{i,j} \ A_{i,j} + B_{i,j} \right)

\text{instantiation} \text{ matrix} :: (\text{plus, type, type}) \text{ plus}
\text{begin}
\text{definition} \text{ plus-matrix-def}: A + B \equiv \text{matrix-plus} A B
\text{instance} ..
\text{end}

\text{lemma} \text{ plus-matrix-def} \left[ \text{simp} \right]:
\quad \text{Matrix} A + \text{Matrix} B = \text{Matrix} \left( \lambda_{i,j} \ A_{i,j} + B_{i,j} \right)
\quad \text{by} (\text{simp add: plus-matrix-def})

\text{instantiation} \text{ sqmatrix} :: (\text{plus, type}) \text{ plus}
\text{begin}
\text{definition} \text{ plus-sqmatrix-def}:
\quad A + B \equiv \text{sqmatrix-of-matrix} (\text{matrix-of-sqmatrix} A + \text{matrix-of-sqmatrix} B)
\text{instance} ..
\text{end}

\text{lemma} \text{ plus-sqmatrix-def} \left[ \text{simp} \right]:
\quad \text{SqMatrix} A + \text{SqMatrix} B = \text{SqMatrix} \left( \lambda_{i,j} \ A_{i,j} + B_{i,j} \right)
\quad \text{by} (\text{simp add: plus-sqmatrix-def})

\text{lemma} \text{ matrix-add-0-right} \left[ \text{simp} \right]:
\quad A + 0 = (A :: (a::monoid-add, 'm, 'n) \text{ matrix})
\quad \text{by} (\text{cases } A, \text{ simp add: zero-matrix-def})

\text{lemma} \text{ matrix-add-0-left} \left[ \text{simp} \right]:
\quad 0 + A = (A :: (a::monoid-add, 'm, 'n) \text{ matrix})
\quad \text{by} (\text{cases } A, \text{ simp add: zero-matrix-def})

\text{lemma} \text{ matrix-add-commute} \left[ \text{simp} \right]:
\quad (A :: (a::ab-semigroup-add, 'm, 'n) \text{ matrix}) + B = B + A
\quad \text{by} (\text{cases } A, \text{ cases } B, \text{ simp add: add.commute})
lemma matrix-add-assoc:
\[(A::(\text{'a::semigroup-add,}'m,'n)\text{ matrix}) + B + C = A + (B + C)\]
by (cases A, cases B, cases C, simp add: add.assoc)

lemma matrix-add-left-commute [simp]:
\[(A::(\text{'a::ab-semigroup-add,}'m,'n)\text{ matrix}) + (B + C) = B + (A + C)\]
by (metis matrix-add-assoc matrix-add-commute)

lemma sqmatrix-add-0-right [simp]:
\[A + 0 = (A::(\text{'a::monoid-add,}'m)\text{ sqmatrix})\]
by (cases A, simp add: zero-sqmatrix-def)

lemma sqmatrix-add-0-left [simp]:
\[0 + A = (A::(\text{'a::monoid-add,}'m)\text{ sqmatrix})\]
by (cases A, simp add: zero-sqmatrix-def)

lemma sqmatrix-add-commute [simp]:
\[(A::(\text{'a::ab-semigroup-add,}'m)\text{ sqmatrix}) + B = B + A\]
by (cases A, cases B, simp add: add.commute)

lemma sqmatrix-add-assoc:
\[(A::(\text{'a::semigroup-add,}'m)\text{ sqmatrix}) + B + C = A + (B + C)\]
by (cases A, cases B, cases C, simp add: add.assoc)

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

5.4 Order (via Addition)

instantiation matrix :: (plus,type,type) plus-ord
begin
definition less-eq-matrix-def:
\[(A::(\text{'a,'b,'c} \text{ matrix}) \leq B \equiv A + B = B)\]
definition less-matrix-def:
\[(A::(\text{'a,'b,'c} \text{ matrix}) < B \equiv A \leq B \land A \neq B)\]

instance
proof
  fix A B :: (\text{'a,'b,'c} \text{ matrix})
  show \(A \leq B \iff A + B = B\)
  by (metis less-eq-matrix-def)
  show \(A < B \iff A \leq B \land A \neq B\)
  by (metis less-matrix-def)
qued
end

instantiation sqmatrix :: (plus,type) plus-ord
begin
\textbf{Definition} \textit{less-eq-sqmatrix-def}:
\[(A::(\'a, 'b) sqmatrix) \leq B \equiv A + B = B\]

\textbf{Definition} \textit{less-sqmatrix-def}:
\[(A::(\'a, 'b) sqmatrix) < B \equiv A \leq B \land A \neq B\]

\textbf{Instance Proof}
\begin{itemize}
\item \textit{fix} \(A B::(\'a, 'b) sqmatrix\)
\item \textit{show} \(A \leq B \iff A + B = B\)
  \textit{by} (metis \textit{less-eq-sqmatrix-def})
\item \textit{show} \(A < B \iff A \leq B \land A \neq B\)
  \textit{by} (metis \textit{less-sqmatrix-def})
\end{itemize}
\textit{qed}
\end{itemize}

\textbf{5.5 Matrix Multiplication}

\textbf{fun matrix-times}::(\(\'a::\{\text{comm-monoid-add, times}\}, \'m, 'k\) matrix \Rightarrow (\(\'a::\{\text{comm-monoid-add, times}\}, \'k\) matrix)

\textbf{where}
\[(\text{matrix-times} (\text{Matrix } A) (\text{Matrix } B)) = \text{Matrix} (\lambda i j. \text{setsum} (\lambda k. A i k \ast B k j)) (\text{UNIV}::'k \text{ atMost set})\]

\textbf{notation} \textit{matrix-times} (infixl \(\ast_M\) 70)

\textbf{instantiation} \textit{sqmatrix}::(\{\text{comm-monoid-add, times}\}, \text{type}) \textit{times}

\textbf{begin}
\textbf{definition} \textit{times-sqmatrix-def}:
\[A \ast B = \text{sqmatrix-of-matrix} (\text{matrix-of-sqmatrix } A \ast_M \text{matrix-of-sqmatrix } B)\]
\textbf{instance} ..
\end{itemize}

\textbf{lemma} \textit{times-sqmatrix-def'} [simp]:
\[\text{SqMatrix } A \ast \text{SqMatrix } B = \text{SqMatrix} (\lambda i j. \text{setsum} (\lambda k. A i k \ast B k j)) (\text{UNIV}::'k \text{ atMost set})\]
\textit{by} (simp add: \textit{times-sqmatrix-def})

\textbf{lemma} \textit{matrix-mult-0-right} [simp]:
\[(A::(\'a::\{\text{comm-monoid-add, mult-zero}\}, 'm, 'n) matrix) \ast_M 0 = 0\]
\textit{by} (cases A, simp add: \textit{zero-matrix-def})

\textbf{lemma} \textit{matrix-mult-0-left} [simp]:
\[0 \ast_M (A::(\'a::\{\text{comm-monoid-add, mult-zero}\}, 'm, 'n) matrix) = 0\]
\textit{by} (cases A, simp add: \textit{zero-matrix-def})

\textbf{lemma} \textit{setsum-delta-r-0} [simp]:
\[\{ \text{finite } S; j \notin S \} \Rightarrow (\sum_{k \in S} f k \ast (if k = j then 1 else (0::'b::\{\text{semiring-0, monoid-mult}\}))) = 0\]
\textit{by} (induct \(S\) rule: finite-induct, auto)

32
\textbf{lemma} \texttt{setsum-delta-r-1 [simp]}:
\[
[ \text{finite } S; j \in S ] \implies (\sum_{k \in S} f * (if k = j \text{ then } 1 \text{ else } (0::\text{semiring-0,monoid-mult}))) = f \ j
\]
\textbf{by} (induct \texttt{S} rule: \texttt{finite-induct, auto})

\textbf{lemma} \texttt{matrix-mult-1-right [simp]}:
\[
(A::\langle a::\text{semiring-0,monoid-mult}, m::\text{len}, n::\text{len} \rangle \text{ matrix}) * _M 1 = A
\]
\textbf{by} (cases \texttt{A}, \texttt{simp add: one-matrix-def})

\textbf{lemma} \texttt{setsum-delta-l-0 [simp]}:
\[
[ \text{finite } S; i \notin S ] \implies (\sum_{k \in S} (if i = k \text{ then } 1 \text{ else } (0::\text{semiring-0,monoid-mult}))) * f \ k \ j = 0
\]
\textbf{by} (induct \texttt{S} rule: \texttt{finite-induct, auto})

\textbf{lemma} \texttt{setsum-delta-l-1 [simp]}:
\[
[ \text{finite } S; i \in S ] \implies (\sum_{k \in S} (if i = k \text{ then } 1 \text{ else } (0::\text{semiring-0,monoid-mult}))) * f \ k \ j = f \ i \ j
\]
\textbf{by} (induct \texttt{S} rule: \texttt{finite-induct, auto})

\textbf{lemma} \texttt{matrix-mult-1-left [simp]}:
\[
1 * _M (A::\langle a::\text{semiring-0,monoid-mult}, m::\text{len}, n::\text{len} \rangle \text{ matrix}) = A
\]
\textbf{by} (cases \texttt{A}, \texttt{simp add: one-matrix-def})

\textbf{lemma} \texttt{matrix-mult-assoc}:
\[
(A::\langle a::\text{semiring-0,monoid-mult}, m, n \rangle \text{ matrix}) * _M B * _M C = A * _M (B * _M C)
\]
\textbf{apply} (cases \texttt{A})
\textbf{apply} (cases \texttt{B})
\textbf{apply} (cases \texttt{C})
\textbf{apply} (simp add: \texttt{setsum-left-distrib setsum-right-distrib mult.assoc})
\textbf{apply} (subst \texttt{setsum.commute})
\textbf{apply} (rule \texttt{refl})
\textbf{done}

\textbf{lemma} \texttt{matrix-mult-distrib-left}:
\[
(A::\langle a::\text{comm-monoid-add,semiring}, m, n::\text{len} \rangle \text{ matrix}) * _M (B + C) = A * _M (B + A * _M C)
\]
\textbf{by} (cases \texttt{A}, \texttt{cases B}, \texttt{cases C}, \texttt{simp add: distrib-left setsum.distrib})

\textbf{lemma} \texttt{matrix-mult-distrib-right}:
\[
((A::\langle a::\text{comm-monoid-add,semiring}, m, n::\text{len} \rangle \text{ matrix}) + B) * _M C = A * _M (C + B * _M C)
\]
\textbf{by} (cases \texttt{A}, \texttt{cases B}, \texttt{cases C}, \texttt{simp add: distrib-right setsum.distrib})

\textbf{lemma} \texttt{sqmatrix-mult-0-right [simp]}:
\[
(A::\langle a::\text{comm-monoid-add,mult-zero}, m \rangle \text{ sqmatrix}) * 0 = 0
\]
\textbf{by} (cases \texttt{A}, \texttt{simp add: zero-sqmatrix-def})

\textbf{lemma} \texttt{sqmatrix-mult-0-left [simp]}:
\[
0 * (A::\langle a::\text{comm-monoid-add,mult-zero}, m \rangle \text{ sqmatrix}) = 0
\]

33
by (cases A, simp add: zero-sqmatrix-def)

lemma sqmatrix-mult-1-right [simp]:
(A::{'a::{semiring-0,monoid-mult},'m::len} sqmatrix) * 1 = A
by (cases A, simp add: one-sqmatrix-def)

lemma sqmatrix-mult-1-left [simp]:
1 * (A::{'a::{semiring-0,monoid-mult},'m::len} sqmatrix) = A
by (cases A, simp add: one-sqmatrix-def)

lemma sqmatrix-mult-assoc:
(A::{'a::{semiring-0,monoid-mult},'m}) sqmatrix) * B * C = A * (B * C)
apply (cases A)
apply (cases B)
apply (cases C)
apply (simp add: setsum-left-distrib setsum-right-distrib mult.assoc)
apply (subst setsum.commute)
apply (rule refl)
done

lemma sqmatrix-mult-distrib-left:
(A::{'a::{comm-monoid-add,semiring},'m::len} sqmatrix) * (B + C) = A * B + A * C
by (cases A, cases B, cases C, simp add: distrib-left setsum.distrib)

lemma sqmatrix-mult-distrib-right:
((A::{'a::{comm-monoid-add,semiring},'m::len} sqmatrix) + B) * C = A * C + B * C
by (cases A, cases B, cases C, simp add: distrib-right setsum.distrib)

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
  fix a :: 'a
  show 0 + a = a
  by (fact add-zerol)
qed

subclass (in semiring-one-zero) semiring-0
proof
  fix a :: 'a
  show 0 * a = 0
  by (fact annil)
  show a * 0 = 0
  by (fact annir)
qed
subclass (in ab-near-semiring-one) monoid-mult ..

instantiation sqmatrix :: (dioid-one-zero,len) dioid-one-zero
begin
instance
proof
  fix A B C :: ('a, 'b) sqmatrix
  show A + B + C = A + (B + C)
    by (fact sqmatrix-add-assoc)
  show A + B = B + A
    by (fact sqmatrix-add-commute)
  show A * B * C = A * (B * C)
    by (fact sqmatrix-mult-assoc)
  show (A + B) * C = A * C + B * C
    by (fact sqmatrix-mult-distrib-right)
  show 1 * A = A
    by (fact sqmatrix-mult-1-left)
  show A * 1 = A
    by (fact sqmatrix-mult-1-right)
  show 0 + A = A
    by (fact sqmatrix-add-0-left)
  show 0 * A = 0
    by (fact sqmatrix-mult-0-left)
  show A * 0 = 0
    by (fact sqmatrix-mult-0-right)
  show A + A = A
    by (cases A, simp)
  show A * (B + C) = A * B + A * C
    by (fact sqmatrix-mult-distrib-left)
  qed
end

5.7 Kleene Star for Matrices

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

end

6 Kleene Algebras

theory Kleene-Algebra
imports Dioid
begin

nitpick-params [timeout = 120]
6.1 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 theory 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.

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

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

lemma star-ref: $1 \leq x^*$
  by (metis add-lub star-unfoldl)

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

lemma star-plus-one [simp]: $1 + x^* = x^*$
by (metis less-eq-def star-ref)

lemma star-ll: $x \cdot x^* \leq x^*$
  by (metis add-lub star-unfoldl)

lemma $x^* \cdot x \leq x^*$
  nitpick [expect=genuine] — 3-element counterexample
  oops

lemma $x \cdot x^* = x^*$
  nitpick [expect=genuine] — 2-element counterexample
  oops
```
Next we show that starred elements are transitive.

**lemma** star-trans-eq [simp]: \(x^* \cdot x^* = x^*

**proof** (rule antisym) — this splits an equation into two inequalities

- have \(x^* + x \cdot x^* \leq x^*

- by (metis add-lub eq-refl star-1l)

- thus \(x^* \cdot x^* \leq x^*

- by (metis star-inductl)

**next show** \(x^* \leq x^* \cdot x^*

- by (metis mult-isor mult-onel star-ref)

**qed**

**lemma** star-trans: \(x^* \cdot x^* \leq x^*

- by (metis eq-refl star-trans-eq)

We now derive variants of the star induction axiom.

**lemma** star-inductl-var: \(x \cdot y \leq y \longrightarrow x^* \cdot y \leq y

**proof**

- assume \(x \cdot y \leq y

- hence \(y + x \cdot y \leq y

- by (metis add-lub eq-refl)

- thus \(x^* \cdot y \leq y

- by (metis star-inductl)

**qed**

**lemma** star-inductl-var-equiv: \(x \cdot y \leq y \iff x^* \cdot y \leq y

**proof**

- assume \(x \cdot y \leq y

- thus \(x^* \cdot y \leq y

- by (metis star-inductl-var)

**next**

- assume \(x^* \cdot y \leq y

- hence \(x^* \cdot y = y

- by (metis less-def less-le-trans mult-isor mult-onel star-ref)

- also have \(x \cdot y = x \cdot x^* \cdot y

- by (metis calculation mult.assoc)

- moreover have \(... \leq x^* \cdot y

- by (metis mult-isor star-1l)

- ultimately show \(x \cdot y \leq y

- by auto

**qed**

**lemma** star-inductl-var-eq: \(x \cdot y = y \longrightarrow x^* \cdot y \leq y

- by (metis eq-iff star-inductl-var)

**lemma** star-inductl-var-eq2: \(y = x \cdot y \longrightarrow y = x^* \cdot y

**proof**

- assume \(y = x \cdot y

- also have \(y \leq x^* \cdot y

- by (metis mult-isor mult-onel star-ref)
thus $y = x^\star \cdot y$
  by (metis calculation eq-iff star-inductl-var-eq)
qed

lemma $y = x \cdot y \iff y = x^\star \cdot y$
  nitpick [expect=genuine] — 2-element counterexample
oops

lemma $x^\star \cdot z \leq y \implies z + x \cdot y \leq y$
  nitpick [expect=genuine] — 3-element counterexample
oops

lemma star-inductl-one: $1 + x \cdot y \leq y \implies x^\star \leq y$
  by (metis mult-oner star-inductl)

lemma star-inductl-star: $x \cdot y^\star \leq y^\star \implies x^\star \leq y^\star$
  by (metis add-lub star-inductl-one star-ref)

lemma star-inductl-eq: $z + x \cdot y = y \implies x^\star \cdot z \leq y$
  by (metis eq-iff star-inductl)

We now prove two facts related to 1.

lemma star-subid: $x \leq 1 \implies x^\star = 1$
proof
  assume $x \leq 1$
  hence $1 + x \cdot 1 \leq 1$
    by (metis add-lub eq-refl mult-oner)
  hence $x^\star \leq 1$
    by (metis mult-oner star-inductl)
  hence $x^\star = 1$
    by (metis eq-iff star-ref)
thus $x^\star = 1$
  by (metis calculation order-trans star-inductl-star)
qed

lemma star-one [simp]: $1^\star = 1$
  by (metis eq-iff star-subid)

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
  have $x \cdot (x + y)^\star \leq (x + y) \cdot (x + y)^\star$
    by (metis add-ub1 mult-isor)
  also have ... $\leq (x + y)^\star$
    by (metis star-1l)
  thus $\theta$thesis
    by (metis calculation order-trans star-inductl-star)
qed

lemma star-subdist-var: $x^\star + y^\star \leq (x + y)^\star$
by (metis add.commute add-lub star-subdist)

**lemma** star-iso: \( x \leq y \rightarrow x^* \leq y^* \)
by (metis less-eq-def star-subdist)

We now prove some more simple properties.

**lemma** star-invol [simp]: \((x^*)^* = x^*
proof (rule antisym)
  have \( x^* \cdot x^* = x^* \)
    by (fact star-trans-eq)
  thus \((x^*)^* \leq x^* \)
    by (metis order-refl star-inductl-star)
  have\((x^*)^* \cdot (x^*)^* \leq (x^*)^* \)
    by (fact star-trans)
  hence \( x \cdot (x^*)^* \leq (x^*)^* \)
    by (metis star-inductl-var-equiv)
  thus \( x^* \leq (x^*)^* \)
    by (metis star-inductl-star)
qed

**lemma** star2 [simp]: \((1 + x)^* = x^*
proof (rule antisym)
  show \( x^* \leq (1 + x)^* \)
    by (metis add-comm star-subdist)
  have \( x^* + x \cdot x^* \leq x^* \)
    by (metis add-lub eq-refl star-1l)
  hence \( (1 + x) \cdot x^* \leq x^* \)
    by (metis mult-onel distrib-right')
  thus \( (1 + x)^* \leq x^* \)
    by (metis star-inductl-star)
qed

**lemma** 1 + x^* \cdot x \leq x^*
  nitpick [expect=genuine] — 3-element counterexample
  oops

**lemma** x \leq x^*
  nitpick [expect=genuine] — 3-element counterexample
  oops

**lemma** x^* \cdot x \leq x^*
  nitpick [expect=genuine] — 3-element counterexample
  oops

**lemma** 1 + x \cdot x^* = x^*
  nitpick [expect=genuine] — 4-element counterexample
  oops

**lemma** x \cdot z \leq z \cdot y \rightarrow x^* \cdot z \leq z \cdot y^*

39
The following facts express inductive conditions that are used to show that
\((x + y)^*\) is the greatest term that can be built from \(x\) and \(y\).

**Lemma** sum-star-closure: \(x \leq z^* \land y \leq z^* \implies x + y \leq z^*\)

by (metis add-lub)

**Lemma** prod-star-closure: \(x \leq z^* \land y \leq z^* \implies x \cdot y \leq z^*\)

proof

assume \(assm: x \leq z^* \land y \leq z^*\)

hence \(y + z^* \cdot z^* \leq z^*\)

by (metis add-lub eq-refl star-trans-eq)

hence \(z^* \cdot y \leq z^*\)

by (metis star-inductl star-invol)

also have \(x \cdot y \leq z^* \cdot y\)

by (metis assm mult-isol)

thus \(x \cdot y \leq z^*\)

by (metis calculation order-trans)

qed

**Lemma** star-star-closure: \(x^* \leq z^* \implies (x^*)^* \leq z^*\)

by (metis star-invol)

**Lemma** star-closed-unfold: \(x^* = x \implies x = I + x \cdot x\)

by (metis star-plus-one star-trans-eq)

**Lemma** \(x^* = x \iff x = I + x \cdot x\)

nitpick [expect=genuine] — 3-element counterexample

oops

end

### 6.2 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: \(x \leq x^*\)

proof

have \(x \leq x \cdot x^*\)

by (metis mult-oner mult-isol star-ref)

thus \(?thesis\)

by (metis order-trans star-1l)

qed

We now prove a right star unfold law.

**Lemma** star-1r: \(x^* \cdot x \leq x^*\)
proof

have \( x + x \cdot x^* \leq x^* \)
  by (metis add-lub star-1l star-ext)

thus \( x \leq x^* \)
  by (metis star-inductl)

qed

lemma star-unfoldr: \( 1 + x^* \cdot x \leq x^* \)
  by (metis star-ref)

lemma \( 1 + x^* \cdot x = x^* \)
  by (metis star-invol star-ref)

nitpick [expect=genuine] — 4-element counterexample

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

lemma star-sim1: \( \exists z \cdot x \leq x \cdot z \leq y \cdot z \leq z \cdot y^* \)

proof

assume \( x \cdot z \leq z \cdot y \)

hence \( x \cdot z \cdot y \leq z \cdot y \cdot y^* \)
  by (metis mult-isor)

also have \( \ldots \leq z \cdot y^* \)
  by (metis mult-assoc mult-isol star-1l)

hence \( z + x \cdot z \cdot y \leq z \cdot y^* \)
  by (metis calculation order-trans add-lub mult-isol mult-oner star-ref)

thus \( x^* \cdot z \leq z \cdot y^* \)
  by (metis mult-assoc star-inductl)

qed

The next lemma is used in omega algebras to prove, for instance, Bachmair and Dershowitz's separation of termination theorem [2]. The property at the left-hand side of the equivalence is known as quasicommutation.

lemma quasicomm-var: \( y \cdot x \leq x \cdot (x + y)^* \iff y^* \cdot x \leq x \cdot (x + y)^* \)

proof

assume \( y \cdot x \leq x \cdot (x + y)^* \)

thus \( y^* \cdot x \leq x \cdot (x + y)^* \)
  by (metis star-invol star-sim1)

next

assume \( y^* \cdot x \leq x \cdot (x + y)^* \)

thus \( y \cdot x \leq x \cdot (x + y)^* \)
  by (metis mult-isol order-trans star-ext)

qed

lemma star-slide1: \( (x \cdot y)^* \cdot x \leq x \cdot (y \cdot x)^* \)
  by (metis eq-iff mult-assoc star-sim1)

lemma \( (x \cdot y)^* \cdot x = x \cdot (y \cdot x)^* \)
  by (metis eq-iff mult-assoc star-sim1)

nitpick [expect=genuine] — 3-element counterexample

oops
lemma star-slide-var1: $x^* \cdot x \leq x \cdot x^*$
  by (metis eq-refl star-sim1)

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

lemma star-unfoldl-eq [simp]: $1 + x \cdot x^* = x^*$
proof (rule antisym)
  show $1 + x \cdot x^* \leq x^*$
    by (fact star-unfoldl)
  have $1 + x \cdot (1 + x \cdot x^*) \leq 1 + x \cdot x^*$
    by (metis add-iso-var eq-refl mult-isol star-unfoldl)
  thus $x^* \leq 1 + x \cdot x^*$
    by (metis star-inductl-one)
qed

lemma $1 + x^* \cdot x = x^*$
  nitpick [expect=genuine] — 4-element counterexample
  oops

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

lemma star-rtc1-eq [simp]: $1 + x + x^* \cdot x^* = x^*$
proof —
  have $1 + x + x^* \cdot x^* = 1 + x + x^*$
    by (metis star-trans-eq)
  also have $... = x + x^*$
    by (metis add.commute add.left-commute star-plus-one)
  thus $?thesis$
    by (metis calculation less-eq-def star-ext)
qed

lemma star-rtc1: $1 + x + x^* \cdot x^* \leq x^*$
  by (metis eq-refl star-rtc1-eq)

lemma star-rtc2: $1 + x \cdot x \leq x \iff x = x^*$
  by (metis antisym star-ext star-inductl-one star-plus-one star-trans-eq)

lemma star-rtc3: $1 + x \cdot x = x \iff x = x^*$
  by (metis order-refl star-plus-one star-rtc2 star-trans-eq)

lemma star-rtc-least: $1 + x + y \cdot y \leq y \rightarrow x^* \leq y$
proof
  assume $1 + x + y \cdot y \leq y$
  hence $1 + x \cdot y \leq y$
    by (metis add-lub less-eq-def distrib-right')
  thus $x^* \leq y$
    by (metis star-inductl-one)
qed
lemma star-rtc-least-eq: \(1 + x + y \cdot y = y \rightarrow x^* \leq y\)  
by (metis eq-refl star-rtc-least)

lemma 1 + x + y \cdot y \leq y \iff x^* \leq y
nitpick [expect=genuine] — 3-element counterexample

The next lemmas are again related to closure conditions

lemma star-subdist-var-1: \(x \leq (x + y)^*\)  
by (metis add-lub star-ext)

lemma star-subdist-var-2: \(x \cdot y \leq (x + y)^*\)  
by (metis add-lub prod-star-closure star-ext)

lemma star-subdist-var-3: \(x^* \cdot y^* \leq (x + y)^*\)  
by (metis add-comm prod-star-closure star-subdist)

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

lemma star-denest [simp]: \((x + y)^* = (x^* \cdot y^*)^*\)  
proof (rule antisym)  
have \(x + y \leq x^* \cdot y^*\)  
  by (metis add-lub-var mult-isol-var mult-onel mult-oner star-ext star-ref)  
thus \((x + y)^* \leq (x^* \cdot y^*)^*\)  
  by (metis star-iso)  
have \(x^* \cdot y^* \leq (x + y)^*\)  
  by (metis star-subdist-var-3)  
thus \((x^* \cdot y^*)^* \leq (x + y)^*\)  
  by (metis star-invol star-iso)  
qed

lemma star-sum-var [simp]: \((x^* + y^*)^* = (x + y)^*\)  
by (metis star-denest star-invol)

lemma star-denest-var [simp]: \((x + y)^* = x^* \cdot (y \cdot x^*)^*\)  
proof (rule antisym)  
have \(1 \leq x^* \cdot (y \cdot x^*)^*\)  
  by (metis add-lub mult-onel star-unfoldl-eq substll)  
moreover have \(x \cdot x^* \cdot (y \cdot x^*)^* \leq x^* \cdot (y \cdot x^*)^*\)  
  by (metis mult-isor star-1l)  
moreover have \(y \cdot x^* \cdot (y \cdot x^*)^* \leq x^* \cdot (y \cdot x^*)^*\)  
  by (metis mult-isol-var mult-onel star-1l star-ref)  
hence \(1 + (x + y) \cdot x^* \cdot (y \cdot x^*)^* \leq x^* \cdot (y \cdot x^*)^*\)  
  by (metis calculation add-lub-var distrib-right')  
thus \((x + y)^* \leq x^* \cdot (y \cdot x^*)^*\)  
  by (metis mult.assoc mult-onel star-distribl)  
have \((y \cdot x^*)^* \leq (y^* \cdot x^*)^*\)  
  by (metis mult-isor star-ext star-iso)  
moreover have \(\ldots = (x + y)^*\)
by (metis add.commute star-denest)
moreover have $x^* \leq (x + y)^*$
  by (metis star-subdist)
thus $x^* \cdot (y \cdot x^*)^* \leq (x + y)^*$
  by (metis calculation prod-star-closure)
qed

lemma star-denest-var-2: $(x^* \cdot y^*)^* = x^* \cdot (y \cdot x^*)^*$
  by (metis star-denest-star-denest-var)

lemma star-denest-var-3 [simp]: $x^* \cdot (y^* \cdot x^*)^* = (x^* \cdot y^*)^*$
  by (metis star-denest-star-denest-var-star-invol)

lemma star-denest-var-4 [simp]: $(y^* \cdot x^*)^* = (x^* \cdot y^*)^*$
  by (metis add-comm star-denest)

lemma star-denest-var-5: $x^* \cdot (y \cdot x^*)^* = y^* \cdot (x \cdot y^*)^*$
  by (metis add-comm star-denest-var)

lemma star-denest-var-6 [simp]: $(x^* \cdot (y \cdot x^*)^*)^* = (x^* \cdot y^*)^*$
  by (metis add-comm star-denest-var-2)

lemma star-denest-var-7 [simp]: $(x^* \cdot (y^* \cdot x^*)^*)^* = (x^* \cdot (y^*)^*)^*$
  by (metis add-comm star-denest-var-3)

proof (rule antisym)
  have $(x + y)^* \cdot x^* \cdot y^* \leq (x + y)^* \cdot (x^* \cdot y^*)^*$
    by (metis mult-isol mult.assoc star-ext)
  thus $(x + y)^* \cdot x^* \cdot y^* \leq (x + y)^*$
    by (metis star-denest star-trans-eq)
  have $1 \leq (x + y)^* \cdot x^* \cdot y^*$
    by (metis mult-isol-var mult-oner star-ref)
  moreover have $(x + y)^* \cdot x^* \cdot y^* \leq (x + y)^* \cdot x^* \cdot y^*$
    by (metis mult-isol star-1l)
  moreover have $1 + (x + y)^* \cdot x^* \cdot y^* \leq (x + y)^* \cdot x^* \cdot y^*$
    by (metis add-lub calculation)
  thus $(x + y)^* \leq (x + y)^* \cdot x^* \cdot y^*$
    by (metis mult.assoc star-inductl-one)
qed

lemma star-denest-var-8 [simp]: $x^* \cdot y^* \cdot (x^* \cdot y^*)^* = (x^* \cdot y^*)^*$
  by (metis mult assoc star-denest-var-2 star-invol)

lemma star-denest-var-9 [simp]: $(x^* \cdot y^*)^* \cdot x^* \cdot y^* = (x^* \cdot y^*)^*$
  by (metis star-denest-star-denest-var-7)

The following statements are well known from term rewriting. They are all
variants of the Church-Rosser theorem in Kleene algebra [25]. But first we
prove a law relating two confluence properties.
lemma confluence-var: \( y \cdot x \leq x \cdot y \leftrightarrow y \cdot x \leq x \cdot y \)
proof
  assume \( y \cdot x \leq x \cdot y \)
  thus \( y \cdot x \leq x \cdot y \)
    by (metis star-invol star-sim1)
next
  assume \( y \cdot x \leq x \cdot y \)
  thus \( y \cdot x \leq x \cdot y \)
    by (metis dual mult-isol order-trans star-ext)
qed

lemma church-rosser: \( y \cdot x \leq x \cdot y \rightarrow (x + y) = x \cdot y \)
proof
  assume \( y \cdot x \leq x \cdot y \)
  hence \( x \cdot y \cdot x \cdot y \leq x \cdot y \cdot y \cdot y \)
    by (metis mult-isol mult-assoc)
  also have \( \leq x \cdot y \)
    by (metis eq-refl mult-associ mult)
  also have \( 1 \leq x \cdot y \)
    by (metis add-lub mult-oner star-unfoldl-eq subdistl)
  hence \( (x \cdot y) \cdot x \cdot y \leq x \cdot y \)
    by (metis add-lub-var calculation mult-assoc)
  hence \( x \cdot y \cdot x \cdot y \leq x \cdot y \)
    by (metis calculation mult-assoc star-denest-var-9 star-inductl-var-equiv)
  moreover have \( (x + y) = x \cdot y \)
    by (metis calculation star-denest)
  thus \( (x + y) = x \cdot y \)
    by (metis eq-iff star-subdist-var-3)
qed

lemma church-rosser-var: \( y \cdot x \leq x \cdot y \rightarrow (x + y) = x \cdot y \)
proof
  by (metis church-rosser confluence-var)

lemma church-rosser-to-confluence: \( (x + y) \leq x \cdot y \rightarrow y \cdot x \leq x \cdot y \)
proof
  by (metis add-comm eq-iff star-subdist-var-3)

lemma church-rosser-equiv: \( y \cdot x \leq x \cdot y \rightarrow (x + y) = x \cdot y \)
proof
  by (metis church-rosser church-rosser-to-confluence eq-iff)

lemma confluence-to-local-confluence: \( y \cdot x \leq x \cdot y \rightarrow y \cdot x \leq x \cdot y \)
proof
  by (metis add.commute church-rosser star-subdist-var-2)

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

lemma sup-id-star1: \( 1 \leq x \rightarrow x \cdot x = x \)
proof
  assume \( 1 \leq x \)
  hence \( x \cdot x \leq x \)
    by (metis mult-isol mult-onel)

The next lemmas relate the reflexive transitive closure and the transitive closure.
thus $x \cdot x^* = x^*$
  by (metis eq-iff star-Il)

qed

lemma sup-id-star2: $1 \leq x \quad \rightarrow \quad x^* \cdot x = x^*$
  by (metis eq-iff mult-isol mult-oner star-1r)

lemma $1 + x^* \cdot x = x^*$
  nitpick [expect=genuine] — 4-element counterexample
  oops

lemma $(x \cdot y)^* \cdot x = x \cdot (y \cdot x)^*$
  nitpick [expect=genuine] — 3-element counterexample
  oops

lemma $x \cdot x = x^* \quad \rightarrow \quad x^* \cdot 1 + x$
  nitpick [expect=genuine] — 4-element counterexample
  oops

end

6.3 Left Kleene Algebras

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

In left Kleene algebras the non-fact $z + y \cdot x \leq y \quad \rightarrow \quad z \cdot x^* \leq y$ 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 [19].

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

lemma star-unfoldr-eq [simp]: $1 + x^* \cdot x = x^*$
proof (rule antisym)
  show $1 + x^* \cdot x \leq x^*$
    by (metis star-unfoldr)
  have $1 + x \cdot (1 + x^* \cdot x) = 1 + (1 + x \cdot x^*) \cdot x$
    by (metis distrib-right mult.assoc mult-onel mult-oner distrib-left)
  also have $\ldots = 1 + x^* \cdot x$
    by (metis star-unfoldl-eq)
  thus $x^* \leq 1 + x^* \cdot x$
    by (metis calculation eq-refl star-ductl-one)
qed

The following more complex unfold law has been used as an axiom, called prodstar, by Conway [6].

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

by (metis mult.assoc star-unfoldr-eq)

thus \((x \cdot y)^* \leq 1 + x \cdot (y \cdot x)^* \cdot y\)
  by (metis add-iso-var mult-isor order-refl star-slide1)

have \(1 + x \cdot (y \cdot x)^* \cdot y \leq 1 + x \cdot y \cdot (x \cdot y)^*\)
  by (metis add-iso-var eq-refl mult.assoc mult-isol star-slide1)

thus \(1 + x \cdot (y \cdot x)^* \cdot y \leq (x \cdot y)^*\)
  by (metis star-unfoldl-eq)

qed

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

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

**proof**

- have \(x \cdot (y \cdot x)^* = x \cdot (1 + y \cdot (x \cdot y)^* \cdot x)\)
  by (metis star-prod-unfold)

- also have \(\ldots = (1 + x \cdot y \cdot (x \cdot y)^*) \cdot x\)
  by (metis distrib-right mult.assoc mult-onel mult-oner distrib-left)

thus ?thesis
  by (metis calculation star-unfoldl-eq)

qed

**lemma** star-slide-var: \(x^* \cdot x = x \cdot x^*\)

by (metis mult-onel mult-oner star-slide)

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

by (metis star-denest star-denest-var-3 star-denest-var-4 star-plus-one star-slide)

The following law shows how starred sums can be unfolded.

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

**proof**

- have \((x + y)^* = x^* \cdot (y \cdot x^*)^*\)
  by (metis star-denest-var)

- also have \(\ldots = x^* \cdot (1 + y \cdot x^* \cdot (y \cdot x^*)^*)\)
  by (metis star-unfoldl-eq)

- also have \(\ldots = x^* \cdot (1 + y \cdot (x + y)^*)\)
  by (metis mult.assoc star-denest-var)

thus ?thesis
  by (metis mult.assoc mult-oner distrib-left calculation)

qed

The following property appears in process algebra.

**lemma** troeger [simp]: \((x + y)^* \cdot z = x^* \cdot (y \cdot (x+y)^* \cdot z + z)\)

using [[(metis-verbose=false)]]

— Theorem opp_mult_def is not actually required for metis to find a proof, but
  (interestingly enough) it considerably speeds up the proof search. We suppress the
  “unused theorem” warning that metis would generate in verbose mode.

by (metis add.commute distrib-left distrib-right mult.assoc mult-onel mult-oner
  opp-mult-def star-sum-unfold)
The following properties are related to a property from propositional dy-
namic logic which has been attributed to Albert Meyer [17]. Here we prove
it as a theorem of Kleene algebra.

**Lemma star-square**: \( (x \cdot x)^* \leq x^* \)

**Proof**

- **have** \( x \cdot x \cdot x^* \leq x^* \)
  - by \( \text{metis mult.assoc prod-star-closure star-1l star-ext} \)
- **thus** ?thesis
  - by \( \text{metis star-inductl-star} \)

**Qed**

**Lemma meyer-1 [simp]**: \( (1 + x) \cdot (x \cdot x)^* = x^* \)

**Proof** \( \text{(rule antisym)} \)

**have** \( 1 \leq (1 + x) \cdot (x \cdot x)^* \)
  - by \( \text{metis add-lub mult-oner star-unfoldl-eq subdistl} \)

**also have** \( x \cdot (1 + x) \cdot (x \cdot x)^* = x \cdot (x \cdot x)^* + x \cdot (x \cdot x)^* \)
  - by \( \text{metis distrib-right mult-oner distrib-left} \)

**moreover have** \( \ldots \leq x \cdot (x \cdot x)^* + (x \cdot x)^* \)
  - by \( \text{metis add-iso-var le-less star-1l} \)

**moreover have** \( \ldots \leq (1 + x) \cdot (x \cdot x)^* \)
  - by \( \text{metis add.commute eq-iff distrib-right mult-onel} \)

**hence** \( 1 + x \cdot (1 + x) \cdot (x \cdot x)^* \leq (1 + x) \cdot (x \cdot x)^* \)
  - by \( \text{metis add.commute add-lub-var calculation distrib-right mult.assoc mult-onel} \)

**thus** \( x^* \leq (1 + x) \cdot (x \cdot x)^* \)
  - by \( \text{metis mult.assoc star-inductl-one} \)

**show** \( (1 + x) \cdot (x \cdot x)^* \leq x^* \)
  - by \( \text{metis prod-star-closure star2 star-ext star-square} \)

**Qed**

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

**Lemma tc**: \( x \cdot x \leq x \rightarrow x^* \cdot x = x \)

**Proof**

**assume** \( x \cdot x \leq x \)

**hence** \( x + x \cdot x \leq x \)
  - by \( \text{metis add-lub eq-refl} \)

**hence** \( x^* \cdot x \leq x \)
  - by \( \text{metis star-inductl} \)

**thus** \( x^* \cdot x = x \)
  - by \( \text{metis multi-isol multi-oner star-ref star-slide-var eq-iff} \)

**Qed**

**Lemma tc-eq**: \( x \cdot x = x \rightarrow x^* \cdot x = x \)

by \( \text{metis order-refl tc} \)

The next fact has been used by Boffa [4] to axiomatise the equational theory
of regular expressions.

**Lemma boffa-var**: \( x \cdot x \leq x \rightarrow x^* = 1 + x \)
proof
  assume $x \cdot x \leq x$
  also have $x^* = 1 + x^* \cdot x$
    by (metis star-unfoldr-eq)
  thus $x^* = 1 + x$
    by (metis calculation tc)
qed

lemma boffa: $x \cdot x = x \rightarrow x^* = 1 + x$
  by (metis boffa-var eq-iff)

end

6.4 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 [28].

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

lemma star-zero [simp]: $0^* = 1$
  by (metis add-zerol less-eq-def star-subid)

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

6.5 Pre-Kleene Algebras

Pre-Kleene algebras are essentially probabilistic Kleene algebras [22]. 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: $z + y \cdot (x + 1) \leq y \rightarrow z \cdot x^* \leq y$

6.6 Kleene Algebras

Finally, here come the Kleene algebras à la Kozen [?]. 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: $z + y \cdot x \leq y \rightarrow z \cdot x^* \leq y$
begin

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**

fix x y z :: 'a
show (x ⊙ y) ⊙ z = x ⊙ (y ⊙ z)
  by (metis mult_assoc opp-mult-def)
show (x + y) ⊙ z = x ⊙ z + y ⊙ z
  by (metis opp-mult-def distrib-left)
show 1 ⊙ x = x
  by (metis mult-oner opp-mult-def)
show 0 ⊙ x = x
  by (metis mult-onel opp-mult-def)
show 0 + x = x
  by (fact add-zerol)
show 0 ⊙ x = 0
  by (metis annir opp-mult-def)
show x ⊙ 0 = 0
  by (metis annil opp-mult-def)
show x + x = x
  by (fact add-idem)
show x ⊙ (y + z) = x ⊙ y + x ⊙ z
  by (metis distrib-right opp-mult-def)
show z ⊙ x ≤ z ⊙ (x + y)
  by (metis mult-isor opp-mult-def order-prop)
show 1 + x ⊙ x* ≤ x*
  by (metis opp-mult-def order-refl star-slide-var star-unfoldl-eq)
show z + x ⊙ y ≤ y → x* ⊙ z ≤ y
  by (metis opp-mult-def star-inductr)
show z + y ⊙ x ≤ y → z ⊙ x* ≤ y
  by (metis opp-mult-def star-inductl)
qed

**lemma** star-inductr-var: y · x ≤ y → y · x* ≤ y
  by (metis add-lub order-refl star-inductr)

**lemma** star-inductr-var-equiv: y · x ≤ y ↔ y · x* ≤ y
  by (metis order-trans mult-isol star-ext star-inductr-var)

The following law could be immediate from star_sim1 if we had proper (technical support for) duality for Kleene algebras.

**lemma** star-sim2: z · x ≤ y · z → z · x* ≤ y* · z

**proof**

from dual-kleene-algebra have class.left-pre-kleene-algebra op+ op⊙ 1 op≤ op< star

unfolding class.kleene-algebra-def class.left-kleene-algebra-zero-def

50
Next we prove two independence properties. In relational Kleene algebras, for instance, $x \cdot y = (0::a)$ means that relation $x$ does not lead into states where relation $y$ can be executed.

**Lemma independence1**: $x \cdot y = 0 \rightarrow x^* \cdot y = y$

**Proof**

- Assume $x \cdot y = 0$
- Also have $x^* \cdot y = y + x^* \cdot x \cdot y$
  - By (metis distrib-right mult-one star-unfoldr-eq)
- Thus $x^* \cdot y = y$
  - By (metis add-zero-r annir calculation mult.assoc)

**QED**

**Lemma independence2**: $x \cdot y = 0 \rightarrow x \cdot y^* = x$

**Proof**

- Let $?t = x \cdot (x + y)^*$
- Assume $y \cdot x \leq {?t + y}$
- Also have $(?t + y) \cdot x = {?t \cdot x + y \cdot x}$
  - By (metis distrib-right)
- Moreover have $\ldots \leq {?t \cdot x + {?t + y}$

The following lemma is important for a separation of termination theorem by Doornbos, Backhouse and van der Woude [9]. The property at the left-hand side has been baptised lazy commutation by Nachum Dershowitz [7].

**Lemma lazycomm-var**: $y \cdot x \leq x \cdot (x + y)^* + y \leftrightarrow y \cdot x^* \leq x \cdot (x + y)^* + y$

**Proof**

- Let $?t = x \cdot (x + y)^*$
- Assume $y \cdot x \leq {?t + y}$
- Also have $(?t + y) \cdot x = {?t \cdot x + y \cdot x}$
  - By (metis distrib-right)
- Moreover have $\ldots \leq {?t \cdot x + {?t + y}$

51
by (metis add-iso-var calculation le-less add.assoc)
moreover have ... ≤ ?t + y
  by (metis add-iso-var add-lub-var mult.assoc mult-isol order-refl prod-star-closure
star-subdist-var-1)
hence y + (?t + y) · x ≤ ?t + y
  by (metis add.commute add-lub-var add-ub1 calculation less-eq-def mult.assoc
distrib-left star-subdist-var-1 star-trans-eq)
thus y · x* ≤ x · (x + y)* + y
  by (metis star-inductr)
next
assume y · x* ≤ x · (x + y)* + y
thus y · x ≤ x · (x + y)* + y
  by (metis mult-isol order-trans star-ext)
qed

Arden’s rule in language theory says that if a language \( X \) does not contain the empty word, then the language equation \( Y = X \cdot Y + Z \) has the unique solution \( Y = X^* \cdot Z \). We prove a variant of this rule, that the Kleene algebra equation \( y = x \cdot y + z \) has the unique solution \( y = x^* \cdot z \), from a different algebraic condition [26].

lemma arden-var: (\( \forall y \cdot v \cdot y \leq x \cdot y + v \rightarrow y \leq x^* \cdot v \) \( \rightarrow z = x \cdot z + w \rightarrow z = x^* \cdot w \))
  by (metis add-comm eq-iff star-inductl-eq)

lemma (\( \forall x \cdot y \cdot y \leq x \cdot y \rightarrow y = 0 \) \( \rightarrow y \leq x \cdot y + z \rightarrow y \leq x^* \cdot z \))
  by (metis eq-refl mult-onel)

end

We finish with some properties on (multiplicatively) commutative Kleene algebras. A chapter in Conway’s book [6] is devoted to this topic.

class commutative-kleene-algebra = kleene-algebra +
  assumes mutl-comm: \( x \cdot y = y \cdot x \)
begin

lemma conway-c3 [simp]: \((x + y)^* = x^* \cdot y^* \)
  by (metis bubble-sort eq-refl mult-comm)

lemma conway-c4 [simp]: \((x^* \cdot y)^* = 1 + x^* \cdot y^* \cdot y \)
  by (metis conway-c3 star-denest-var star-prod-unfold)

lemma cka-1: \((x \cdot y)^* \leq x^* \cdot y^* \)
  by (metis conway-c3 star-invol star-iso star-subdist-var-2)

lemma cka-2 [simp]: \(x^* \cdot (x^* \cdot y)^* = x^* \cdot y^* \)
  by (metis conway-c3 mult-comm star-denest-var)
lemma conway-c4-var [simp]: \((x^* \cdot y^*)^* = x^* \cdot y^*\)
  by (metis conway-c3 star-invol)

lemma conway-c2 [simp]: \((x \cdot y)^* \cdot (x^* + y^*) = x^* \cdot y^*\)
proof (rule antisym)
  show \((x \cdot y)^* \cdot (x^* + y^*) \leq x^* \cdot y^*\)
    by (metis cka-1 conway-c3 prod-star-closure star-ext star-sum-var)
  have \(x \cdot (x \cdot y)^* \cdot (x^* + y^*) = x \cdot (x \cdot y)^* \cdot (x^* + 1 + y \cdot y^*)\)
    by (metis add.commute add.left-commute star-unfoldl-eq)
  also have \(\ldots = x \cdot (x \cdot y)^* \cdot (x^* + y \cdot y^*)\)
    by (metis add.commute star-plus-one)
  also have \(\ldots = (x \cdot y)^* \cdot x \cdot x^* + (x \cdot y)^* \cdot x \cdot y \cdot y^*\)
    by (metis distrib-left mult.assoc mult-comm)
  also have \(\ldots \leq (x \cdot y)^* \cdot x^* + (x \cdot y)^* \cdot x \cdot y \cdot y^*\)
    by (metis star-1l add-iso mult-isol mult.assoc)
  also have \(\ldots \leq (x \cdot y)^* \cdot x^* + (x \cdot y)^* \cdot y^*\)
    by (metis add-isovar conway-c3 mult.assoc mult-comm order-refl prod-star-closure star-subdist-var-1)
  also have \(\ldots = (x \cdot y)^* \cdot (x^* + y^*)\)
    by (metis distrib-right mult-comm)
  finally have \(x \cdot (x \cdot y)^* \cdot (x^* + y^*) \leq (x \cdot y)^* \cdot (x^* + y^*)\)
    by (metis add-lub anir mult-oner star-sim2 star-zero zero-least)
  hence \(y^* + x \cdot (x \cdot y)^* \cdot (x^* + y^*) \leq (x \cdot y)^* \cdot (x^* + y^*)\)
    by (metis add-lub calculation)
  thus \(x^* \cdot y^* \leq (x \cdot y)^* \cdot (x^* + y^*)\)
    by (metis mult.assoc star-inductl)
qed

end
end

7 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.
7.1 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: $z + x \cdot y \leq y \rightarrow (x \cdot n) \cdot z \leq y$
proof (induct n)
case 0 show ?case by (metis add-lub mult-onel power-0)
case Suc thus ?case by (auto, metis add-lub mult.assoc mult-isol order-trans)
qed

lemma power-inductr: $z \cdot y \cdot x \leq y \rightarrow z \cdot (x \cdot n) \leq y$
proof (induct n)
case 0 show ?case by (metis add-lub mult-oner power-0)
case Suc
{
  fix n
  assume $z + y \cdot x \leq y \rightarrow z \cdot x \cdot n \leq y$
  and $z + y \cdot x \leq y$
  hence $z \cdot x \cdot n \leq y$
    by auto
  also have $z \cdot x \cdot Suc n = z \cdot x \cdot x \cdot n$
    by (metis mult.assoc power-Suc)
  moreover have ... $= (z \cdot x \cdot n) \cdot x$
    by (metis mult.assoc power-commutes)
  moreover have ... $\leq y \cdot x$
    by (metis calculation(1) mult-isor)
  moreover have ... $\leq y$
    by (metis (z + y \cdot x \leq y) add-lub)
  ultimately have $z \cdot x \cdot Suc n \leq y$ by auto
}
  thus ?case by (metis Suc)
qed

end

7.2 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: $(\bigcup n. f n) = f 0 \cup (\bigcup n. f (Suc n))$
by \textit{auto} (metis not0-implies-Suc)

\textbf{instantiation} \textit{set} :: (monoid-mult) kleene-algebra
\textbf{begin}

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

\textbf{lemma} \textit{star-elim}: \(x \in X^* \iff (\exists k. x \in X^k)\)
\textbf{by} (simp add: star-def)

\textbf{lemma} \textit{star-contl}: \(X \cdot Y^* = (\bigcup n. X \cdot Y^n)\)
\textbf{by} (auto simp add: star-elim c-prod-def)

\textbf{lemma} \textit{star-contr}: \(X^* \cdot Y = (\bigcup n. X^n \cdot Y)\)
\textbf{by} (auto simp add: star-elim c-prod-def)

\textbf{instance} \textbf{proof}
\textbf{fix} \(X Y Z :: \text{''a set}\)
\textbf{show} \(1 + X \cdot X^* \subseteq X^*\)
\textbf{proof} –
\textbf{have} \(1 + X \cdot X^* = (X^0) \cup (\bigcup n. X^n \cdot \text{Suc }n)\)
\textbf{by} (auto simp add: star-def c-prod-def plus-set-def one-set-def)
\textbf{also have} \(\ldots = (\bigcup n. X^n)\)
\textbf{by} (metis Un-0-Suc)
\textbf{also have} \(\ldots = X^*\)
\textbf{by} (simp only: star-def)
\textbf{finally show} \(?\text{thesis}\)
\textbf{by} (metis subset-refl)
\textbf{qed}

\textbf{show} \(Z + X \cdot Y \subseteq Y \longrightarrow X^* \cdot Z \subseteq Y\)
\textbf{by} (simp add: star-contr SUP-le-iff, metis power-inductr)
\textbf{show} \(Z + Y \cdot X \subseteq Y \longrightarrow Z \cdot X^* \subseteq Y\)
\textbf{by} (simp add: star-contl SUP-le-iff, metis power-inductl)
\textbf{qed}

\end

7.3 Language Kleene Algebras

We now specialise this fact to languages.

\textbf{interpretation} \textit{lan-kleene-algebra}: kleene-algebra \(op + \text{op} \cdot 1::\text{''a lan op} \subseteq \text{op} \subseteq \text{star }\).

7.4 Regular Languages

\ldots and further to regular languages. For the sake of simplicity we just copy in the axiomatisation of regular expressions by Krauss and Nipkow [21].
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)*

typedef 'a reg-lan = range lang :: 'a lan set
  by auto

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
  by (metis (hide-lams, no-types) image-iff lang.simps(6) rangeI)

lift-definition zero-reg-lan :: 'a reg-lan
  is 0
  by (metis lang.simps(1) rangeI)

lift-definition one-reg-lan :: 'a reg-lan
  is 1
  by (metis lang.simps(2) rangeI)

lift-definition less-eq-reg-lan :: 'a reg-lan ⇒ 'a reg-lan ⇒ bool
  is less-eq

lift-definition less-reg-lan :: 'a reg-lan ⇒ 'a reg-lan ⇒ bool
  is less

lift-definition plus-reg-lan :: 'a reg-lan ⇒ 'a reg-lan ⇒ 'a reg-lan
  is plus
  by (metis (hide-lams, no-types) image-iff lang.simps(4) rangeI)

lift-definition times-reg-lan :: 'a reg-lan ⇒ 'a reg-lan ⇒ 'a reg-lan
instance

proof

  fix x y z :: 'a reg-lan
  show x + y + z = x + (y + z)
    by transfer (metis join-semilattice-class.add-assoc)
  show x + y = y + x
    by transfer (metis join-semilattice-class.add-comm)
  show x · y · z = x · (y · z)
    by transfer (metis semigroup-mult-class.mult-assoc)
  show (x + y) · z = x · z + y · z
    by transfer (metis semiring-class.distrib-right)
  show 1 · x = x
    by transfer (metis monoid-mult-class.mult-1-left)
  show x · 1 = x
    by transfer (metis monoid-mult-class.mult-1-right)
  show 0 + x = x
    by transfer (metis join-semilattice-zero-class.add-zero-l)
  show 0 · x = 0
    by transfer (metis ab-near-semiring-one-zero-class.annil)
  show x · 0 = 0
    by transfer (metis ab-near-semiring-one-zero-class.annir)
  show x ≤ y ←→ x + y = y
    by transfer (metis plus-ord-class.less-eq-def)
  show x < y ←→ x ≤ y ∧ x ≠ y
    by transfer (metis plus-ord-class.less-def)
  show x + x = x
    by transfer (metis join-semilattice-class.add-idem)
  show x · (y + z) = x · y + x · z
    by transfer (metis semiring-class.distrib-left)
  show z · x ≤ z · (x + y)
    by transfer (metis pre-dioid-class.subdistl)
  show 1 + x · x* ≤ x*
    by transfer (metis star-unfoldl)
  show z + x · y ≤ y → x* · z ≤ y
    by transfer (metis star-inductl)
  show z + y · x ≤ y → z · x* ≤ y
    by transfer (metis star-inductr)

qed

end

interpretation reg-lan-kleene-algebra: kleene-algebra op + op · 1::'a reg-lan 0 op ≤ op < star ..
7.5 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** (induct `n`)

- **Case 0**: Show `?case` by (metis rel-dioid.power-0 relpow.simps(1))
- **Case Suc**: Thus `?case` by (metis rel-dioid.power-Suc2 relpow.simps(2))

**Qed**

**Lemma**: `rel-star-def: X^* = (∪ n. rel-dioid.power X n)`

**Proof**

- `simp add: power-is-relpow rtrancl-is-UN-relpow`

**Qed**

**Lemma**: `rel-star-contl: X O Y^* = (∪ n. X O rel-dioid.power Y n)`

**Proof**

- `metis rel-star-def relcomp-UNION-distrib`

**Qed**

**Lemma**: `rel-star-contr: X^* O Y = (∪ n. (rel-dioid.power X n) O Y)`

**Proof**

- `metis rel-star-def relcomp-UNION-distrib2`

**Qed**

**Interpretation**: `rel-kleene-algebra: kleene-algebra op ∪ op O Id {}` `op ⊆ op ⊂ rtrancl`

**Proof**

- Fix `x y z :: 'a rel`
- Show `Id ∪ x O x^* ⊆ x^*` by (metis order-refl r-comp-rtrancl-eq rtrancl-unfold)
- Show `z ∪ x O y ⊆ y ⟹ x^* O z ⊆ y` by (simp only: rel-star-contr, metis (lifting) SUP-le-iff rel-dioid.power-inductl)
- Show `z ∪ y O x ⊆ x^* ⟹ z O x^* ⊆ y` by (simp only: rel-star-contl, metis (lifting) SUP-le-iff rel-dioid.power-inductr)

**Qed**

7.6 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** (simp add: t-star-def)

**Qed**

**Lemma**: `t-star-contl: t-prod X (t-star Y) = (∪ n. t-prod X (trace-dioid.power Y n))`

**Proof** (auto simp add: t-star-elim t-prod-def)

**Qed**
lemma t-star-contr: t-prod (t-star X) Y = (∪ n. t-prod (trace-dioid.power X n) Y)
by (auto simp add: t-star-elim t-prod-def)

interpretation trace-kleene-algebra: kleene-algebra op ∪ t-prod t-one t-zero op ⊆ op ⊂ t-star
proof
  fix X Y Z :: ('a, 'b) trace set
  show t-one ∪ t-prod X (t-star X) ⊆ t-star X
  proof
    have t-one ∪ t-prod X (t-star X) = (trace-dioid.power X 0) ∪ (∪ n. trace-dioid.power X n)
    by (auto simp add: t-star-def t-prod-def)
    also have ... = (∪ n. trace-dioid.power X n)
    by (metis Un-0-Suc)
    also have ... = t-star X
    by (metis t-star-def)
    finally show ?thesis
    by (metis subset-refl)
  qed

  show Z ∪ t-prod X Y ⊆ Y → t-prod (t-star X) Z ⊆ Y
  by (simp only: ball-UNIV t-star-contr SUP-le-iff (metis trace-dioid.power-inductl))

  show Z ∪ t-prod Y X ⊆ Y → t-prod Z (t-star X) ⊆ Y
  by (simp only: ball-UNIV t-star-contl SUP-le-iff (metis trace-dioid.power-inductr))
qed

7.7 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 ċ (∃ n. x ∈ path-dioid.power X n)
by (simp add: p-star-def)

lemma p-star-contl: p-prod X (p-star Y) = (∪ n. p-prod X (path-dioid.power Y n))
apply (auto simp add: p-prod-def p-star-elim)
  apply (metis p-fusion.simps(1))
  apply metis
  apply (metis p-fusion.simps(1) p-star-elim)
apply (metis p-star-elim)
done

lemma p-star-contr: p-prod (p-star X) Y = (∪ n. p-prod (path-dioid.power X n) Y)
apply (auto simp add: p-prod-def p-star-elim)
  apply (metis p-fusion.simps(1))
apply metis
apply (metis p-fusion.simps(1) p-star-elim)
apply (metis p-star-elim)
done

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

proof
  fix X Y Z :: 'a path set
  show p-one ∪ p-prod X (p-star X) ⊆ p-star X
  proof
    have p-one ∪ p-prod X (p-star X) = (path-dioid.power X 0) ∪ (∪ n. path-dioid.power X n)
    by (auto simp add: p-star-def p-prod-def)
    also have ... = (∪ n. path-dioid.power X n)
    by (metis Un-0-Suc)
    also have ... = p-star X
    by (metis p-star-def)
    finally show ?thesis
    by (metis subset-refl)
  qed
show Z ∪ p-prod X Y ⊆ Y −→ p-prod (p-star X) Z ⊆ Y
by (simp only: ball-UNIV p-star-contr SUP-le-iff (metis path-dioid.power-inductl))
show Z ∪ p-prod Y X ⊆ Y −→ p-prod Z (p-star X) ⊆ Y
by (simp only: ball-UNIV p-star-contl SUP-le-iff (metis path-dioid.power-inductr))
qed

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)
by (simp add: pp-star-def)

lemma pp-star-contl: pp-prod X (pp-star Y) = (∪ n. pp-prod X (ppath-dioid.power Y n))
by (auto simp add: pp-prod-def pp-star-elim)

lemma pp-star-contr: pp-prod (pp-star X) Y = (∪ n. pp-prod (ppath-dioid.power X n) Y)
by (auto simp add: pp-prod-def pp-star-elim)

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

proof
  fix X Y Z :: 'a ppath set
  show pp-one ∪ pp-prod X (pp-star X) ⊆ pp-star X
  proof
    have pp-one ∪ pp-prod X (pp-star X) = (ppath-dioid.power X 0) ∪ (∪ n. path-dioid.power X n)
    by (auto simp add: p-star-def)
  qed

  also have ...
  by (simp only: ball-UNIV pp-star-contr pp-star-elim)
  also have ...
  by (metis path-dioid.power-induct)
  also have ...
  by (metis p-star-def)
  finally show ?thesis
  by (metis subset-refl)
  qed

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)
by (simp add: pp-star-def)

lemma pp-star-contl: pp-prod X (pp-star Y) = (∪ n. pp-prod X (ppath-dioid.power Y n))
by (auto simp add: pp-prod-def pp-star-elim)

lemma pp-star-contr: pp-prod (pp-star X) Y = (∪ n. pp-prod (ppath-dioid.power X n) Y)
by (auto simp add: pp-prod-def pp-star-elim)

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

proof
  fix X Y Z :: 'a ppath set
  show pp-one ∪ pp-prod X (pp-star X) ⊆ pp-star X
  proof
    have pp-one ∪ pp-prod X (pp-star X) = (ppath-dioid.power X 0) ∪ (∪ n. path-dioid.power X n)
    by (auto simp add: p-star-def)
  qed

  also have ...
  by (simp only: ball-UNIV pp-star-contr pp-star-elim)
  also have ...
  by (metis path-dioid.power-induct)
  also have ...
  by (metis p-star-def)
  finally show ?thesis
  by (metis subset-refl)
  qed
7.8 The Distributive Lattice Kleene Algebra

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

\[
\text{definition (in bounded-distributive-lattice) bdl-star :: } 'a \Rightarrow 'a \text{ where}
\]

\[
\text{bdl-star x = top}
\]

\[
\text{sublocale bounded-distributive-lattice \subseteq kleene-algebra sup inf top bot less-eq less}
\]

\[
bdl-star
\]

\[
\text{proof}
\]

\[
\text{fix } x y z :: 'a
\]

\[
\text{show sup top (inf x (bdl-star x)) \leq bdl-star x}
\]

\[
\text{by (simp add: bdl-star-def)}
\]

\[
\text{show sup z (inf x y) \leq y \rightarrow inf (bdl-star x) z \leq y}
\]

\[
\text{by (simp add: bdl-star-def)}
\]

\[
\text{show sup z (inf y x) \leq y \rightarrow inf z (bdl-star x) \leq y}
\]

\[
\text{by (simp add: bdl-star-def)}
\]

7.9 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.

\[
\text{instantiation pnat :: commutative-kleene-algebra}
\]

\[
\text{begin}
\]

\[
\text{definition star-pnat where}
\]

\[
x^* \equiv (1::pnat)
\]
instance
proof
  fix x y z :: pnat
  show 1 + x · x^* ≤ x^*
    by (metis star-pnat-def zero-pnat-top)
  show z + x · y ≤ y −→ x^* · z ≤ y
    by (metis join-semilattice-class.add-lub monoid-mult-class.mult-1-left star-pnat-def)
  show z + y · x ≤ y −→ z · x^* ≤ y
    by (metis join-semilattice-class.add-lub monoid-mult-class.mult-1-right star-pnat-def)
  show x · y = y · x
    unfolding times-pnat-def by (cases x, cases y, simp-all)
qed

end
end

8 Action Algebras

theory Action-Algebra
imports Kleene-Algebra
begin

Action algebras have been defined and discussed in Vaughan Pratt’s paper on Action Logic and Pure Induction [24]. They are expansions of Kleene algebras by operations of left and right residuation. They are interesting, first because most models of Kleene algebras, e.g. relations, traces, paths and languages, possess the residuated structure, and second because, in this setting, the Kleene star can be equationally defined.

Action algebras can be based on residuated semilattices [13], which are interesting in their own right. Many important properties of action algebras already arise at this level.

Here we only prove some basic properties of residuated semilattices and action algebras. A more extensive treatment is left for future work. There is also an obvious duality between proofs for left and right residuation which we do not formalise at this stage.

class residuated-join-semilattice = join-semilattice + semigroup-mult + residual-l-op + residual-r-op +
  assumes residual-l-galois: x ≤ z −→ y ≤ x · y ≤ z
  and residual-r-galois: x · y ≤ z −→ y ≤ x −→ z
begin

We first prove unit and counit laws for residuals, which are also known as cancellation laws.

lemma galois-unitl: x ≤ x · y −→ y
  by (metis eq-refl residual-l-galois)
lemma galois-counitl: \((y \leftarrow x) \cdot x \leq y\)
by (metis eq-refl residual-l-galois)

lemma galois-unitr: \(y \leq x \rightarrow x \cdot y\)
by (metis eq-refl residual-r-galois)

lemma galois-counitr: \(x \cdot (x \rightarrow y) \leq y\)
by (metis eq-refl residual-r-galois)

Next we show that distributivity laws hold (in fact, even distributivity laws
for all existing suprema).

lemma distl: \(x \cdot (y + z) = x \cdot y + x \cdot z\)
proof -
{  
  fix \(w\)
  have \(x \cdot (y + z) \leq w \iff y + z \leq x \rightarrow w\)
    by (metis residual-r-galois)
  also have \(\ldots \iff y \leq x \rightarrow w \land z \leq x \rightarrow w\)
    by (fact add-lub)
  also have \(\ldots \iff x \cdot y \leq w \land x \cdot z \leq w\)
    by (metis residual-r-galois)
  ultimately have \(x \cdot (y + z) \leq w \iff x \cdot y + x \cdot z \leq w\)
    by (metis add-lub)
}  
thus ?thesis
  by (metis eq-iff)
qed

lemma distr: \((x + y) \cdot z = x \cdot z + y \cdot z\)
proof -
{  
  fix \(w\)
  have \((x + y) \cdot z \leq w \iff x + y \leq w \iff z\)
    by (metis residual-l-galois)
  also have \(\ldots \iff x \leq w \iff z \land y \leq w \iff z\)
    by (fact add-lub)
  also have \(\ldots \iff x \cdot z \leq w \land y \cdot z \leq w\)
    by (metis residual-l-galois)
  ultimately have \((x + y) \cdot z \leq w \iff x \cdot z + y \cdot z \leq w\)
    by (metis add-lub)
}  
thus ?thesis
  by (metis eq-iff)
qed

As usual, distributivity implies isotonicity.

lemma mult-isol: \(x \leq y \rightarrow z \cdot x \leq z \cdot y\)
by (metis distl less-eq-def)
lemma mult-isor: \( x \leq y \longrightarrow x \cdot z \leq y \cdot z \)
by (metis distr less-eq-def)

Similarly, the residuals as upper adjoints preserve all existing meets, but we
do not assume that any meets exist in residuated semilattices. However we
can show subdistributivity with respect to residuation.

lemma residual-l-subdist-var: \( x \leftarrow z \leq (x + y) \leftarrow z \)
proof –
{  
  fix \( w \)
  have \( w \leq x \leftarrow z \iff w \cdot z \leq x \)
  by (metis residual-l-galois)
  also have \( \ldots \longrightarrow w \cdot z \leq x + y \)
  by (metis add-ub1 order-trans)
  ultimately have \( w \leq x \leftarrow z \longrightarrow w \leq (x + y) \leftarrow z \)
  by (metis residual-l-galois)
}
thus \( \text{thesis} \)
by (metis eq-refl)
qed

lemma residual-l-subdist: \( (x \leftarrow z) + (y \leftarrow z) \leq (x + y) \leftarrow z \)
by (metis add-comm add-lub residual-l-subdist-var)

lemma residual-r-subdist-var: \( (x \rightarrow y) \leq x \rightarrow (y + z) \)
by (metis add-ub1 galois-counitr order-trans residual-r-galois)

lemma residual-r-subdist: \( (x \rightarrow y) + (x \rightarrow z) \leq x \rightarrow (y + z) \)
by (metis add-comm add-lub residual-r-subdist-var)

As usual, subdistributivity implies isotonicity.

lemma residual-l-isol: \( x \leq y \longrightarrow x \leftarrow z \leq y \leftarrow z \)
by (metis less-eq-def residual-l-subdist-var)

lemma residual-r-isor: \( x \leq y \longrightarrow z \rightarrow x \leq z \rightarrow y \)
by (metis less-eq-def residual-r-subdist-var)

Next, we prove superdistributivity laws for residuation.

lemma residual-l-superdist-var: \( x \leftarrow (y + z) \leq x \leftarrow y \)
proof –
{  
  fix \( w \)
  have \( w \leq x \leftarrow (y + z) \leftarrow w \cdot (y + z) \leq x \)
  by (metis residual-l-galois)
  also have \( \ldots \leftarrow w \cdot y \leq x \land w \cdot z \leq x \)
  by (metis add-lub distl)
  also have \( \ldots \leftarrow w \leq x \leftarrow y \land w \leq x \leftarrow z \)
by (metis residual-l-galois)
finally have $w \leq x \leftarrow (y + z) \rightarrow w \leq x \leftarrow y$
  by simp
}
thus ?thesis
  by (metis eq-refl)
qed

lemma residual-r-superdist-var: $(x + y) \rightarrow z \leq x \rightarrow z$
  by (metis add-lub galois-counitr residual-l-galois residual-r-galois)

The previous proof shows, in fact, that $x \leftarrow (y + z)$ is the infimum of $x \leftarrow y$ and $x \leftarrow z$; but we have no operation to express this fact in action algebra. A dual property holds for right residuation.

As usual, superdistributivity implies antitonicity.

lemma residual-l-antitoner: $x \leq y \rightarrow z \leftarrow y \leq z \leftarrow x$
  by (metis residual-l-superdist-var)

lemma residual-r-antitonel: $x \leq y \rightarrow y \rightarrow z \leq x \rightarrow z$
  by (metis residual-r-superdist-var)

Finally we prove transitivity laws for residuals.

lemma residual-l-trans: $(x \leftarrow y) \cdot (y \leftarrow z) \leq x \leftarrow z$
proof
  have $(x \leftarrow y) \cdot y \leq x$
    by (metis galois-counitl)
  hence $(x \leftarrow y) \cdot (y \leftarrow z) \cdot z \leq x$
    by (metis galois-counitl mult.assoc residual-l-antitoner residual-l-galois)
  thus ?thesis
    by (metis residual-l-galois)
qed

lemma residual-r-trans: $(x \rightarrow y) \cdot (y \rightarrow z) \leq x \rightarrow z$
proof
  have $y \cdot (y \rightarrow z) \leq z$
    by (metis galois-counitr)
  hence $x \cdot (x \rightarrow y) \cdot (y \rightarrow z) \leq z$
    by (metis galois-counitr mult.assoc residual-r-antitonel residual-r-galois)
  thus ?thesis
    by (metis galois-counitle residual-r-galois)
qed

end

We now present an equivalent equational axiomatisation of residuated join semilattices, which is essentially derived from an equational axiomatisation of Galois connections in algebras with sufficient structure. This equivalence
is the basis for establishing the equivalence of the equational axiomatisation of action algebra and that based on Galois connections.

class equation-algebra = join-semilattice + semigroup-mult +
residual-l-op + residual-r-op +
assumes mult-subdist: z · x ≤ z · (x + y)
and mult-subdistr: x · z ≤ (x+y) · z
and right-addition: x → y ≤ x → (y + z)
and right-galois-counit: x · (x → y) ≤ y
and right-galois-unit: y ≤ x → x · y
and left-addition: y ← x ≤ (y + z) ← x
and left-galois-counit: (y ← x) · x ≤ y
and left-galois-unit: y ≤ y · x ← x

begin

lemma residual-l-galois': x · y ≤ z ↔ x ≤ z ← y
proof
  assume x · y ≤ z
  hence (x · y) ← y ≤ z ← y
    by (metis less-eq-def left-addition)
  thus x ≤ z ← y
    by (metis order-trans left-galois-unit)
next
  assume x ≤ z ← y
  hence x · y ≤ (z ← y) · y
    by (metis less-eq-def mult-subdistr)
  thus x · y ≤ z
    by (metis order-trans left-galois-counit)
qed

lemma residual-r-galois': x · y ≤ z ↔ y ≤ x → z
proof
  assume x · y ≤ z
  hence x → (x · y) ≤ x → z
    by (metis less-eq-def right-addition)
  thus y ≤ x → z
    by (metis order-trans right-galois-unit)
next
  assume y ≤ x → z
  hence x · y ≤ x · (x → z)
    by (metis less-eq-def mult-subdistr)
  thus x · y ≤ z
    by (metis order-trans right-galois-counit)
qed

subclass residuated-join-semilattice
  by (unfold-locales, metis residual-l-galois', residual-r-galois')

end
Conversely, every residuated join semilattice satisfies the axioms of equa-
tional residuated join semilattices.

Because the subclass relation must be acyclic in Isabelle, we can only estab-
lish this for the corresponding locales.

\textbf{sublocale residuated-join-semilattice} \subseteq \textbf{equational-residuated-join-semilattice}

\begin{itemize}
  \item by (unfold-locales, metis add-ub1 mult-isol, metis add-ub1 mult-isor, metis residual-r-subdist-var, metis galois-counitr, metis galois-unitr, metis residual-l-subdist-var, metis galois-counitl, metis galois-unitl)
\end{itemize}

We can now define an action algebra as a residuated join semilattice that is
also a dioid. Following Pratt, we also add a star operation that is axiomati-
sed as a reflexive transitive closure operation.

\textbf{class action-algebra} = \textbf{residuated-join-semilattice} + \textbf{dioid-one-zero} + \textbf{star-op} +

\begin{itemize}
  \item assumes star-rtc1: \(1 + x^\star \cdot x^\star + x \leq x^\star\)
  \item and star-rtc2: \(1 + y \cdot y + x \leq y \longrightarrow x^\star \leq y\)
\end{itemize}

\textbf{begin}

We first prove a reflexivity property for residuals.

\textbf{lemma residual-r-refl:} \(1 \leq x \rightarrow x\)

\textbf{proof} –

\begin{itemize}
  \item have \(x \leq x\)
  \item thus \(?\text{thesis}\)
  \item by (metis mult-oner residual-r-galois)
\end{itemize}

\textbf{qed}

\textbf{lemma residual-l-refl:} \(1 \leq x \leftarrow x\)

\textbf{proof} –

\begin{itemize}
  \item have \(x \leq x\)
  \item by auto
  \item thus \(?\text{thesis}\)
  \item by (metis mult-onel residual-l-galois)
\end{itemize}

\textbf{qed}

We now derive pure induction laws for residuals.

\textbf{lemma residual-l-pure-induction:} \((x \leftarrow x)^\star \leq x \leftarrow x\)

\textbf{proof} –

\begin{itemize}
  \item have \(1 + (x \leftarrow x) \cdot (x \leftarrow x) + (x \leftarrow x) \leq (x \leftarrow x)\)
  \item by (metis add-lub eq-iff residual-l-refl residual-l-trans)
  \item thus \(?\text{thesis}\)
  \item by (metis star-rtc2)
\end{itemize}

\textbf{qed}

\textbf{lemma residual-r-pure-induction:} \((x \rightarrow x)^\star \leq x \rightarrow x\)

\textbf{by} (metis add-lub eq-iff residual-r-refl residual-r-trans star-rtc2)

Next we show that every action algebra is a Kleene algebra. First, we derive
the star unfold law and the star induction laws in action algebra. Then we prove a subclass statement.
lemma star-unfoldl: \(1 + x \cdot x^* \leq x^*\)
proof
  have \(x \cdot x^* \leq x^*\)
    by (metis add-lub mult-isor order-trans star-rtc1)
  thus ?thesis
    by (metis add-lub star-rtc1)
qed

lemma star-mon: \(x \leq y \rightarrow x^* \leq y^*\)
proof
  assume \(x \leq y\)
  hence \(x \leq y^*\)
    by (metis add-lub order-trans star-rtc1)
  hence \(1 + x + y^* \cdot y^* \leq y^*\)
    by (metis add-lub star-rtc1)
  thus \(x^* \leq y^*\)
    by (metis add.assoc add.commute star-rtc2)
qed

lemma star-subdist': \(x^* \leq (x + y)^*\)
  by (metis add-ub1 star-mon)

lemma star-inductl: \(z + x \cdot y \leq y \rightarrow x^* \cdot z \leq y\)
proof
  assume \(z + x \cdot y \leq y\)
  also have \(z \leq y\)
    by (metis add-lub calculation)
  moreover have \(x \cdot y \leq y\)
    by (metis add-lub calculation)
  hence \(x \leq y \leftarrow y\)
    by (metis residual-l-galois)
  hence \(x^* \leq (y \leftarrow y)^*\)
    by (metis star-mon)
  hence \(x^* \leq y \leftarrow y\)
    by (metis order-trans residual-l-pure-induction)
  hence \(x^* \cdot y \leq y\)
    by (metis residual-l-galois)
  thus \(x^* \cdot z \leq y\)
    by (metis calculation mult-isol order-trans)
qed

lemma star-inductr: \(z + y \cdot x \leq y \rightarrow z \cdot x^* \leq y\)
proof
  assume \(z + y \cdot x \leq y\)
  also have \(z \leq y\)
    by (metis add-lub calculation)
  moreover have \(y \cdot x \leq y\)
by (metis add-lub calculation)
hence $x \leq y \rightarrow y$
  by (metis residual-r-galois)

hence $x^* \leq (y \rightarrow y)^*$
  by (metis star-mon)

hence $x^* \leq y \rightarrow y$
  by (metis order-trans residual-r-pure-induction)

hence $y \cdot x^* \leq y$
  by (metis residual-r-galois)

thus $z \cdot x^* \leq y$
  by (metis order-trans residual-r-pure-induction)

qed

subclass kleene-algebra
  by (unfold-locales, auto simp add: star-unfoldl star-inductl star-inductr)
end

8.1 Equational Action Algebras

The induction axioms of Kleene algebras are universal Horn formulas. This
is unavoidable, because due to a well known result of Redko, there is no finite
equational axiomatisation for the equational theory of regular expressions.

Action algebras, in contrast, admit a finite equational axiomatization, as
Pratt has shown. We now formalise this result. Consequently, the equational
action algebra axioms, which imply those based on Galois connections, which
in turn imply those of Kleene algebras, are complete with respect to the
equational theory of regular expressions. However, this completeness result
does not account for residuation.

class equational-action-algebra = equational-residuated-join-semilattice + dioid-one-zero
  + star-op +
  assumes star-ax: $1 + x^* \cdot x^* + x \leq x^*$
  and star-subdist: $x^* \leq (x + y)^*$
  and right-pure-induction: $(x \rightarrow x)^* \leq x \rightarrow x$
begin

We now show that the equational axioms of action algebra satisfy those
based on the Galois connections. Since we can use our correspondence be-
tween the two variants of residuated semilattice, it remains to derive the
second reflexive transitive closure axiom for the star, essentially copying
Pratt’s proof step by step. We then prove a subclass statement.

lemma star-rtc-2: $1 + y \cdot y + x \leq y \rightarrow x^* \leq y$
proof
  assume $1 + y \cdot y + x \leq y$
  also have $1 \leq y$
    by (metis add-lub calculation)
  moreover have $x \leq y$

by (metis add-lub calculation)
moreover have \( y \cdot y \leq y \)
  by (metis add-lub calculation)
hence \( y \leq y \rightarrow y \)
  by (metis residual-r-galois)
moreover have \( x \leq y \rightarrow y \)
  by (metis calculation order-trans)
hence \( x^* \leq (y \rightarrow y)^* \)
  by (metis less-eq-def star-subdist)
hence \( x^* \leq y \rightarrow y \)
  by (metis order-trans right-pure-induction)
hence \( y \cdot x^* \leq y \)
  by (metis residual-r-galois)
ultimately show \( x^* \leq y \)
  by (metis mult-isor mult-onel order-trans)
qed

subclass action-algebra
  by (unfold-locales, metis star-ax, metis star-rtc-2)
end

Conversely, every action algebra satisfies the equational axioms of equational action algebras.

Because the subclass relation must be acyclic in Isabelle, we can only establish this for the corresponding locales. Again this proof is based on the residuated semilattice result.

subclass action-algebra \( \subseteq \) equational-action-algebra
  by (unfold-locales, metis star-rtc1, metis star-subdist, metis residual-r-pure-induction)

8.2 Another Variant

Finally we show that Pratt and Kozen’s star axioms generate precisely the same theory.

class action-algebra-var = equational-residuated-join-semilattice + dioid-one-zero
  + star-op +
  assumes star-unfold\(\prime\): \( 1 + x \cdot x^* \leq x^* \)
  and star-inductl\(\prime\): \( z + x \cdot y \leq y \rightarrow x^* \cdot z \leq y \)
  and star-inductr\(\prime\): \( z + y \cdot x \leq y \rightarrow z \cdot x^* \leq y \)
begin

subclass kleene-algebra
  by (unfold-locales, metis star-unfold\(\prime\), metis star-inductl\(\prime\), metis star-inductr\(\prime\))

subclass action-algebra
  by (unfold-locales, metis add.commute less-eq-def order-refl star-ext star-plus-one
    star-trans-eq, metis add.assoc add.commute star-rtc-least)


\textbf{9 Models of Action Algebras}

theory \textit{Action-Algebra-Models}  
\textbf{imports} \textit{Action-Algebra Kleene-Algebra-Models}  
\textbf{begin}

\textbf{9.1 The Powerset Action Algebra over a Monoid}

Here we show that various models of Kleene algebras are also residuated;  
\textit{hence they form action algebras. In each case the main work is to establish  
the residuated lattice structure.}

The interpretation proofs for some of the following models are quite similar.  
\textit{One could, perhaps, abstract out common reasoning.}

\textbf{9.2 The Powerset Action Algebra over a Monoid}

\textbf{instantiation set :: (monoid-mult) residuated-join-semilattice}  
\textbf{begin}

\textbf{definition} residual-r-set where  
\textit{X} \rightarrow \textit{Z} = \bigcup \{\textit{Y}. \textit{X} \cdot \textit{Y} \subseteq \textit{Z}\}

\textbf{definition} residual-l-set where  
\textit{Z} \leftarrow \textit{Y} = \bigcup \{\textit{X}. \textit{X} \cdot \textit{Y} \subseteq \textit{Z}\}

\textbf{instance}  
\textbf{proof}  
\textbf{fix} \textit{X Y Z} :: 'a set  
\textbf{show} \textit{X} \subseteq (\textit{Z} \leftarrow \textit{Y}) \iff \textit{X} \cdot \textit{Y} \subseteq \textit{Z}  
\textbf{proof}  
\textbf{assume} \textit{X} \subseteq \textit{Z} \leftarrow \textit{Y}  
\textbf{hence} \textit{X} \cdot \textit{Y} \subseteq (\textit{Z} \leftarrow \textit{Y}) \cdot \textit{Y}  
\textit{by (metis near-dioid-class.mult-isor)}  
\textbf{also have} \ldots \subseteq \bigcup \{\textit{X}. \textit{X} \cdot \textit{Y} \subseteq \textit{Z}\} \cdot \textit{Y}  
\textit{by (simp add: residual-l-set-def)}  
\textbf{also have} \ldots = \bigcup \{\textit{X} \cdot \textit{Y} | \textit{X}. \textit{X} \cdot \textit{Y} \subseteq \textit{Z}\}  
\textit{by (auto simp only: c-prod-def)}  
\textbf{finally show} \textit{X} \cdot \textit{Y} \subseteq \textit{Z}  
\textit{by auto}  
\textbf{next}

71
assume \( X \cdot Y \subseteq Z \)
hence \( X \subseteq \bigcup \{ X. X \cdot Y \subseteq Z \} \)
by (metis Sup-upper mem-Collect-eq)
thus \( X \subseteq Z \leftarrow Y \)
by (simp add: residual-l-set-def)
qed
show \( X \cdot Y \subseteq Z \leftarrow\rightarrow Y \subseteq (X \rightarrow Z) \)
proof
assume \( Y \subseteq X \rightarrow Z \)
hence \( X \cdot Y \subseteq X \cdot (X \rightarrow Z) \)
by (metis pre-dioid-class.mult-isol)
also have \( \ldots \subseteq X \cdot \bigcup \{ Y. X \cdot Y \subseteq Z \} \)
by (simp add: residual-r-set-def)
also have \( \ldots = \bigcup \{ X \cdot Y \mid Y. X \cdot Y \subseteq Z \} \)
by (auto simp only: c-prod-def)
finally show \( X \cdot Y \subseteq Z \)
by auto
next
assume \( X \cdot Y \subseteq Z \)
hence \( Y \subseteq \bigcup \{ Y. X \cdot Y \subseteq Z \} \)
by (metis Sup-upper mem-Collect-eq)
thus \( Y \subseteq X \rightarrow Z \)
by (simp add: residual-r-set-def)
qed
qed

instantiation set :: (monoid-mult) action-algebra
begin

instance
proof
fix \( x \ y :: 'a \ set \)
show \( 1 + x^* \cdot x^* + x \subseteq x^* \)
by (metis join-semilattice-class.add-lub left-near-kleene-algebra-class.star-plus-one
left-near-kleene-algebra-class.star-trans-eq left-pre-kleene-algebra-class.star-ext subset-refl)
show \( 1 + y \cdot y + x \subseteq y \rightarrow x^* \subseteq y \)
by (metis join-semilattice-class.add-comm join-semilattice-class.add-left-comm
left-pre-kleene-algebra-class.star-rtc-least)
qed

end

9.3 Language Action Algebras

definition limp-lan :: 'a lan ⇒ 'a lan ⇒ 'a lan where
limp-lan \( Z \ Y = \{ x. \forall y \in Y. x \at y \in Z \} \)
**Definition**

\texttt{rimp-lan} :: \texttt{'a lan} ⇒ \texttt{'a lan} ⇒ \texttt{'a lan where}

\texttt{rimp-lan} \texttt{X} \texttt{Z} = \{ y. \forall x \in X. x \circ y \in Z \}

**Interpretation**

\texttt{lan-residuated-join-semilattice}: residuated-join-semilattice \texttt{op + op}

\texttt{⊆ op ⊂ op} \texttt{· limp-lan rimp-lan}

**Proof**

\begin{itemize}
  \item \texttt{fix} \texttt{x y z} :: \texttt{'a lan}
  \item \texttt{show} \texttt{x} \texttt{⊆ limp-lan} \texttt{y} \texttt{←→ x \cdot y} \texttt{⊆ z}
    \texttt{by (auto simp add: c-prod-def limp-lan-def times-list-def)}
  \item \texttt{show} \texttt{x \cdot y} \texttt{⊆ z} \texttt{←→ y} \texttt{⊆ rimp-lan} \texttt{x z}
    \texttt{by (auto simp add: c-prod-def rimp-lan-def times-list-def)}
\end{itemize}

**QED**

**Interpretation**

\texttt{lan-action-algebra}: action-algebra \texttt{op + op \subseteq op \subset op \cdot limp-lan rimp-lan}

**Proof**

\begin{itemize}
  \item \texttt{fix} \texttt{x y} :: \texttt{'a lan}
  \item \texttt{show} \texttt{1 \star x \cdot x} \texttt{⊆ x \star x}
    \texttt{by (metis left-near-kleene-algebra-class.star-plus-one left-near-kleene-algebra-class.star-trans
      left-near-kleene-algebra-class.star-trans-eq left-near-kleene-algebra-class.sum-star-closure
      left-pre-kleene-algebra-class.star-ext)}
  \item \texttt{show} \texttt{1 \star y \cdot y \cdot x} \texttt{⊆ y \cdot x \star x}
    \texttt{by (metis join-semilattice-class.add-lub left-near-kleene-algebra-class.star-iso
      left-pre-kleene-algebra-class.star-rtc2)}
\end{itemize}

**QED**

9.4 Relation Action Algebras

**Definition**

\texttt{rimp-rel} :: \texttt{'a rel} ⇒ \texttt{'a rel} ⇒ \texttt{'a rel where}

\texttt{rimp-rel} \texttt{T} \texttt{S} = \{ (y,x) \mid y x. \forall z. (x,z) \in S \rightarrow (y,z) \in T \}

**Definition**

\texttt{rimp-rel} :: \texttt{'a rel} ⇒ \texttt{'a rel} ⇒ \texttt{'a rel where}

\texttt{rimp-rel} \texttt{R} \texttt{T} = \{ (y,z) \mid y z. \forall x. (x,y) \in R \rightarrow (x,z) \in T \}

**Interpretation**

\texttt{rel-residuated-join-semilattice}: residuated-join-semilattice \texttt{op \cup op}

\texttt{⊆ op \subset op \cdot limp-rel rimp-rel}

**Proof**

\begin{itemize}
  \item \texttt{fix} \texttt{x y z} :: \texttt{'a rel}
  \item \texttt{show} \texttt{x} \texttt{⊆ limp-rel} \texttt{y} \texttt{←→ x \cup y} \texttt{⊆ z}
    \texttt{by (auto simp add: limp-rel-def)}
  \item \texttt{show} \texttt{x \cup y} \texttt{⊆ z} \texttt{←→ y} \texttt{⊆ rimp-rel} \texttt{x z}
    \texttt{by (auto simp add: rimp-rel-def)}
\end{itemize}

**QED**

**Interpretation**

\texttt{rel-action-algebra}: action-algebra \texttt{op \cup op \subseteq op \subset op \cdot limp-rel rimp-rel}

**Proof**

\begin{itemize}
  \item \texttt{fix} \texttt{x y} :: \texttt{'a rel}
  \item \texttt{show} \texttt{Id} \texttt{∪ x \cdot O} \texttt{x \cdot x} \texttt{⊆ x \cdot x}
\end{itemize}
by auto

show \text{Id} \cup y \circ y \cup x \subseteq y \rightarrow x^* \subseteq y
  by (metis le-supE rel-kleene-algebra.star2 rtrancl_mono)

qed

9.5 Trace Action Algebras

definition limp-trace :: \langle'p,'a\rangle trace set \Rightarrow \langle'p,'a\rangle trace set
where
  \text{limp-trace} X Y = \bigcup \{X. \text{t-prod} X Y \subseteq Z\}

definition rimp-trace :: \langle'p,'a\rangle trace set \Rightarrow \langle'p,'a\rangle trace set
where
  \text{rimp-trace} X Z = \bigcup \{Y. \text{t-prod} X Y \subseteq Z\}

interpretation trace-residuated-join-semilattice: residuated-join-semilattice \text{op} \cup \text{op} \subseteq \text{op} \subseteq \text{t-prod} \text{limp-trace} \text{rimp-trace}

proof
  fix X Y Z :: \langle'a,'b\rangle trace set
  show X \subseteq \text{limp-trace} Z Y \iff \text{t-prod} X Y \subseteq Z
  proof
    assume X \subseteq \text{limp-trace} Z Y
    hence \text{t-prod} X Y \subseteq \text{t-prod} (\text{limp-trace} Z Y) Y
      by (metis trace-dioid_mult-isor)
    also have \ldots \subseteq \text{t-prod} (\bigcup \{X. \text{t-prod} X Y \subseteq Z\}) Y
      by (simp add: limp-trace_def)
    also have \ldots = \bigcup \{\text{t-prod} X Y \mid X. \text{t-prod} X Y \subseteq Z\}
      by (auto simp only: t-prod_def)
    finally show \text{t-prod} X Y \subseteq Z
      by auto
  next
    assume \text{t-prod} X Y \subseteq Z
    hence X \subseteq \bigcup \{X. \text{t-prod} X Y \subseteq Z\}
      by (metis Sup_upper mem_Collect_eq)
    thus X \subseteq \text{limp-trace} Z Y
      by (simp add: limp-trace_def)
  qed

  show \text{t-prod} X Y \subseteq Z \iff Y \subseteq \text{rimp-trace} X Z
  proof
    assume \text{t-prod} X Y \subseteq Z
    hence Y \subseteq \bigcup \{Y. \text{t-prod} X Y \subseteq Z\}
      by (metis Sup_upper mem_Collect_eq)
    thus Y \subseteq \text{rimp-trace} X Z
      by (simp add: rimp-trace_def)
  next
    assume Y \subseteq \text{rimp-trace} X Z
    hence \text{t-prod} X Y \subseteq \text{t-prod} X (\text{rimp-trace} X Z)
      by (metis trace-dioid_mult_isor)
    also have \ldots \subseteq \text{t-prod} X (\bigcup \{Y. \text{t-prod} X Y \subseteq Z\})
interpretation trace-action-algebra: action-algebra op \cup \subseteq op \subset t-prod limp-trace
rimp-trace t-one t-zero t-star
proof
fix X Y :: ('a,'b) trace set
show t-one \cup t-prod (t-star X) \cup X \subseteq t-star X
by (metis Un-least order-refl trace-kleene-algebra.star-ext trace-kleene-algebra.star-plus-one trace-kleene-algebra.star-trans-eq)

qed

9.6 Path Action Algebras

We start with paths that include the empty path.

definition limp-path :: 'a path set \Rightarrow 'a path set \Rightarrow 'a path set where
limp-path Z Y = \bigcup \{ X. p-prod X Y \subseteq Z \}

definition rimp-path :: 'a path set \Rightarrow 'a path set \Rightarrow 'a path set where
rimp-path X Z = \bigcup \{ Y. p-prod X Y \subseteq Z \}

interpretation path-residuated-join-semilattice: residuated-join-semilattice op \cup \subseteq op \subset p-prod limp-path rimp-path
proof
fix X Y Z :: 'a path set
show X \subseteq limp-path Z Y \iff p-prod X Y \subseteq Z
proof
assume X \subseteq limp-path Z Y
hence p-prod X Y \subseteq p-prod (limp-path Z Y) Y
by (metis path-dioid.mult-isor)
also have \ldots \subseteq p-prod (\bigcup \{ X. p-prod X Y \subseteq Z \}) Y
by (simp add: limp-path-def)
also have \ldots = \bigcup \{ p-prod X Y \mid X. p-prod X Y \subseteq Z \}
by (auto simp only: p-prod-def)
finally show p-prod X Y \subseteq Z
by auto
next
assume p-prod X Y \subseteq Z
hence X \subseteq \bigcup \{ X. p-prod X Y \subseteq Z \}
by (metis Sup-upper mem-Collect-eq)
thus X \subseteq limp-path Z Y

75
by (simp add: limp-path-def)

cqed

show \( p \cdot \text{prod} \ X \ Y \subseteq \ Z \iff Y \subseteq \text{rimp-path} \ X \ Z \)

proof

assume \( p \cdot \text{prod} \ X \ Y \subseteq \ Z \)
hence \( Y \subseteq \bigcup \{ Y. \ p \cdot \text{prod} \ X \ Y \subseteq \ Z \} \)
by (metis Sup-upper mem-Collect-eq)
thus \( Y \subseteq \text{rimp-path} \ X \ Z \)
by (simp add: limp-path-def)

next

assume \( Y \subseteq \text{rimp-path} \ X \ Z \)
hence \( p \cdot \text{prod} \ X \ Y \subseteq p \cdot \text{prod} \ X (\text{rimp-path} \ X \ Z) \)
by (metis path-dioid.mult-isol)
also have \( \ldots \subseteq \text{rimp-path} X (\bigcup \{ Y. \ p \cdot \text{prod} \ X \ Y \subseteq \ Z \}) \)
by (simp add: limp-path-def)
also have \( \ldots = \bigcup \{ \text{pp-prod} X \ Y \mid Y. \ p \cdot \text{prod} \ X \ Y \subseteq \ Z \} \)
by (auto simp only: p-prod-def)
finally show \( p \cdot \text{prod} \ X \ Y \subseteq \ Z \)
by auto
cqed

cqed

interpretation path-action-algebra: action-algebra op \( \cup \) op \( \subseteq \) op \( \subset \) p-prod limp-path
rimp-path \( \text{p-one} \) \{ \} \text{p-star}

proof

fix \( X \ Y :: \text{'}a \text{ path set} \)
show \( \text{p-one} \cup \text{p-prod} (\text{p-star} \ X) (\text{p-star} \ X) \cup X \subseteq \text{p-star} \ X \)
by (metis Un-least order-refl path-kleene-algebra.star-ext path-kleene-algebra.star-plus-one
path-kleene-algebra.star-trans-eq)
show \( \text{p-one} \cup \text{p-prod} Y \ Y \cup X \subseteq Y \longrightarrow \text{p-star} \ X \subseteq Y \)
by (metis Un-commute le-iff-sup path-dioid.add-lub path-kleene-algebra.baffa-var
path-kleene-algebra.star-subdist)
cqed

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

definition limp-ppath :: \( \text{'}a \text{ path set} \Rightarrow \text{'}a \text{ path set} \Rightarrow \text{'}a \text{ path set} \text{ where} \)
\( \text{limp-ppath} \ Z \ Y = \bigcup \{ X. \ \text{pp-prod} X \ Y \subseteq \ Z \} \)

definition rimp-ppath :: \( \text{'}a \text{ path set} \Rightarrow \text{'}a \text{ path set} \Rightarrow \text{'}a \text{ path set} \text{ where} \)
\( \text{rimp-ppath} \ X \ Z = \bigcup \{ Y. \ \text{pp-prod} X \ Y \subseteq \ Z \} \)

interpretation pp-path-residuated-join-semilattice: residuated-join-semilattice op \( \cup \) op \( \subseteq \) op \( \subset \) pp-prod limp-ppath rimp-ppath

proof

fix \( X \ Y \ Z :: \text{'}a \text{ path set} \)
show \( X \subseteq \text{limp-ppath} \ Z \ Y \iff \text{pp-prod} X \ Y \subseteq \ Z \)
proof

assume \( X \subseteq \text{limp-ppath} \ Z \ Y \)
hence \( \text{pp-prod} X \ Y \subseteq \text{pp-prod} (\text{limp-ppath} \ Z \ Y) \ Y \)
by (metis ppath-dioid.mult-isol)
also have \( \ldots \subseteq \text{pp-prod} (\bigcup \{ X. \text{pp-prod} X Y \subseteq Z\}) \) \( Y \)
by (simp add: limp-ppath-def)
also have \( \ldots = \bigcup \{ \text{pp-prod} X Y \mid X. \text{pp-prod} X Y \subseteq Z\} \)
by (auto simp only: pp-prod-def)
finally show \( \text{pp-prod} X Y \subseteq Z \)
by auto

next
assume \( \text{pp-prod} X Y \subseteq Z \)
hence \( X \subseteq \bigcup \{ X. \text{pp-prod} X Y \subseteq Z\} \)
by (metis Sup-upper mem-Collect-eq)
thus \( X \subseteq \text{limp-ppath} Z Y \)
by (simp add: limp-ppath-def)
qed

show \( \text{pp-prod} X Y \subseteq Z \longleftrightarrow Y \subseteq \text{rimp-ppath} X Z \)
proof
assume \( \text{pp-prod} X Y \subseteq Z \)
hence \( Y \subseteq \bigcup \{ Y. \text{pp-prod} X Y \subseteq Z\} \)
by (metis Sup-upper mem-Collect-eq)
thus \( Y \subseteq \text{rimp-ppath} X Z \)
by (simp add: rimp-ppath-def)
next
assume \( Y \subseteq \text{rimp-ppath} X Z \)
hence \( \text{pp-prod} X Y \subseteq \text{pp-prod} X (\text{rimp-ppath} X Z) \)
by (metis ppath-dioid.mult-isol)
also have \( \ldots \subseteq \text{pp-prod} X (\bigcup \{ Y. \text{pp-prod} X Y \subseteq Z\}) \)
by (simp add: rimp-ppath-def)
also have \( \ldots = \bigcup \{ \text{pp-prod} X Y \mid Y. \text{pp-prod} X Y \subseteq Z\} \)
by (auto simp only: pp-prod-def)
finally show \( \text{pp-prod} X Y \subseteq Z \)
by auto
qed

interpretation ppath-action-algebra: action-algebra op \( \cup \) op \( \subseteq \) op \( \subset \) pp-prod limp-ppath
rimp-ppath pp-one \{\} pp-star
proof
fix \( X Y :: \text{a ppath set} \)
show \( \text{pp-one} \cup \text{pp-prod} (\text{pp-star} X) (\text{pp-star} X) \cup X \subseteq \text{pp-star} X \)
by (metis Un-least order-refl ppath-kleene-algebra.star-ext ppath-kleene-algebra.star-plus-one
ppath-kleene-algebra.star-trans-eq)
show \( \text{pp-one} \cup \text{pp-prod} Y Y \cup X \subseteq Y \longrightarrow \text{pp-star} X \subseteq Y \)
by (metis Un-commute le-iff-sup ppath-dioid.add-lub ppath-kleene-algebra.boffa-var
ppath-kleene-algebra.star-subdist)
qed

9.7 The Min-Plus Action Algebra

instantiation pnat :: minus
fun minus-pnat where
\[(\text{pnat } x) - (\text{pnat } y) = \text{pnat } (x - y)\]
\[x - P\text{Infty} = 1\]
\[P\text{Infty} - (\text{pnat } x) = 0\]

instance ..

end

instantiation pnat :: residuated-join-semilattice
begin

definition residual-r-pnat where
\[(x::\text{pnat}) \to y \equiv y - x\]

definition residual-l-pnat where
\[(y::\text{pnat}) \leftarrow x \equiv y - x\]

instance proof
  fix x y z :: pnat
  show \[x \leq (z \leftarrow y) \leftrightarrow x \cdot y \leq z\]
    by (cases x, cases y, cases z) (auto simp add: plus-pnat-def times-pnat-def residual-l-pnat-def less-eq-pnat-def zero-pnat-def one-pnat-def)
  show \[x \cdot y \leq z \leftrightarrow y \leq (x \to z)\]
    by (cases y, cases x, cases z) (auto simp add: plus-pnat-def times-pnat-def residual-r-pnat-def less-eq-pnat-def zero-pnat-def one-pnat-def)
  qed

end

instantiation pnat :: action-algebra
begin

The Kleene star for type \text{pnat} has already been defined in theory \text{Kleene-Algebra-Models}.

instance proof
  fix x y :: pnat
  show \[1 + x^* \cdot x^* + x \leq x^*\]
    by (metis star-pnat-def zero-pnat-top)
  show \[1 + y \cdot y + x \leq y \rightarrow x^* \leq y\]
    by (metis join-semilattice-class.add-lub star-pnat-def)
  qed

end

end
Omega Algebras

theory Omega-Algebra
imports Kleene-Algebra
begin

Omega algebras [5] extend Kleene algebras by an $\omega$-operation that axiomatizes infinite iteration (just like the Kleene star axiomatizes finite iteration).

10.1 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: $y \leq z + x \cdot y \longrightarrow y \leq x^\omega + x^* \cdot z$
begin

First we prove some variants of the coinduction axiom.

lemma omega-coinduct-var1: $y \leq 1 + x \cdot y \longrightarrow y \leq x^\omega + x^*$
by (metis mult-oner omega-coinduct)

lemma omega-coinduct-var2: $y \leq x \cdot y \longrightarrow y \leq x^\omega$
by (metis add.commute add-zero-l annir omega-coinduct)

lemma omega-coinduct-eq: $y = z + x \cdot y \longrightarrow y \leq x^\omega + x^* \cdot z$
by (metis eq-refl omega-coinduct)

lemma omega-coinduct-eq-var1: $y = 1 + x \cdot y \longrightarrow y \leq x^\omega + x^*$
by (metis eq-refl omega-coinduct-var1)

lemma omega-coinduct-eq-var2: $y = x \cdot y \longrightarrow y \leq x^\omega$
by (metis eq-refl omega-coinduct-var2)

lemma $y = x \cdot y + z \longrightarrow y = x^* \cdot z + x^\omega$
nitpick [expect=genuine] — 2-element counterexample
oops

lemma $y = 1 + x \cdot y \longrightarrow y = x^\omega + x^*$
nitpick [expect=genuine] — 3-element counterexample
oops

lemma $y = x \cdot y \longrightarrow y = x^\omega$
nitpick [expect=genuine] — 2-element counterexample
oops

Next we strengthen the unfold law to an equation.
lemma omega-unfold-eq [simp]: $x \cdot x^\omega = x^\omega$

proof (rule antisym)

have $x \cdot x^\omega \leq x \cdot x^\omega$
  by (metis mult.assoc mult-isol omega-unfold)

thus $x \cdot x^\omega \leq x^\omega$
  by (metis mult.assoc omega-coinduct-var2)

show $x^\omega \leq x \cdot x^\omega$
  by (fact omega-unfold)

qed

lemma omega-unfold-var: $z + x \cdot x^\omega \leq x^\omega + x^\star \cdot z$

by (metis add-lub add-ub1 omega-coinduct omega-unfold-eq)

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

nitpick [expect=genuine] — 4-element counterexample

oops

We now prove subdistributivity and isotonicity of omega.

lemma omega-subdist: $x^\omega \leq (x + y)^\omega$

proof

have $x^\omega \leq (x + y) \cdot x^\omega$
  by (metis add-ub1 mult-isor omega-unfold-eq)

thus ?thesis
  by (metis omega-coinduct-var2)

qed

lemma omega-iso: $x \leq y \rightarrow x^\omega \leq y^\omega$

by (metis less-eq-def omega-subdist)

lemma omega-subdist-var: $x^\omega + y^\omega \leq (x + y)^\omega$

by (metis add.commute add-lub omega-subdist)

lemma zero-omega [simp]: $0^\omega = 0$

by (metis annil omega-unfold-eq)

The next lemma is another variant of omega unfold

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

proof (rule antisym)

have $x \cdot x^\omega \leq x^\omega$
  by (metis eq-refl omega-unfold-eq)

thus $x^* \cdot x^\omega \leq x^\omega$
  by (metis star-inductl-var)

show $x^\omega \leq x^* \cdot x^\omega$
  by (metis star-ref mult-isor mult-onel)

qed

The next lemma says that $1^{\omega}$ is the maximal element of omega algebra.

We therefore baptise it $\top$.

lemma max-element: $x \leq 1^{\omega}$
definition top (⊤) where ⊤ = 1 ω

lemma star-omega-3 [simp]: (x*)ω = ⊤
proof
  have 1 ≤ x*
    by (fact star-ref)
  hence ⊤ ≤ (x*)ω
    by (metis omega-iso top-def)
  thus ?thesis
    by (metis eq-iff max-element top-def)
qed

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

lemma omega-1: xω · y ≤ xω
proof
  have xω · y ≤ x · xω · y
    by (metis eq-refl omega-unfold-eq)
  thus ?thesis
    by (metis mult assoc omega-coinduct-var2)
qed

nitpick [expect=genuine] — 2-element counterexample

oops

lemma omega-sup-id: 1 ≤ y → xω · y = xω
by (metis eq-iff mult-isol mult-oner omega-1)

lemma omega-top [simp]: xω · ⊤ = xω
by (metis max-element omega-sup-id top-def)

lemma supid-omega: 1 ≤ x → xω = ⊤
by (metis eq-iff max-element omega-iso top-def)

lemma xω = ⊤ → 1 ≤ x
nitpick [expect=genuine] — 4-element counterexample

oops

Next we prove a simulation law for the omega operation

lemma omega-simulation: z · x ≤ y · z → z · xω ≤ yω
proof
  assume z · x ≤ y · z
  also have z · xω = z · x · xω
    by (metis mult.assoc omega-unfold-eq)
  moreover have ... ≤ y · z · xω
  by (metis eq-refl mult-onel omega-coinduct-var2)
by (metis mult-isor calculation)
thus \( z \cdot x^\omega \leq y^\omega \)
by (metis calculation mult.assoc omega-coinduct-var2)
qed


lemma \( z \cdot x \leq y \cdot z \rightarrow z \cdot x^\omega \leq y^\omega \cdot z \)
nitpick [expect=genuine] — 4-element counterexample
oops

lemma \( y \cdot z \leq z \cdot x \rightarrow y^\omega \leq z \cdot x^\omega \)
nitpick [expect=genuine] — 2-element counterexample
oops

lemma \( y \cdot z \leq z \cdot x \rightarrow y^\omega \cdot z \leq x^\omega \)
nitpick [expect=genuine] — 4-element counterexample
oops


Next we prove transitivity of omega elements.


lemma omega-trans: \( x^\omega \cdot x^\omega \leq x^\omega \)
by (fact omega-1)


lemma omega-omega: \( (x^\omega)^\omega \leq x^\omega \)
by (metis omega-1 omega-unfold-eq)

The next lemmas are axioms of Wagner’s complete axiomatisation for omega-
regular languages [29], but in a slightly different setting.


lemma wagner-1 [simp]: \( (x \cdot x^*)^\omega = x^\omega \)
proof (rule antisym)
    have \( (x \cdot x^*)^\omega = x \cdot x^* \cdot x \cdot x^* \cdot (x \cdot x^*)^\omega \)
    by (metis mult.assoc omega-unfold-eq)
    also have \( ... = x \cdot x \cdot x^* \cdot x^* \cdot (x \cdot x^*)^\omega \)
    by (metis mult.assoc star-slide-var)
    also have \( ... = x \cdot x \cdot x^* \cdot (x \cdot x^*)^\omega \)
    by (metis mult.assoc star-trans-eq)
    also have \( ... = x \cdot (x \cdot x^*)^\omega \)
    by (metis mult.assoc omega-unfold-eq)
    thus \( (x \cdot x^*)^\omega \leq x^\omega \)
    by (metis calculation eq-refl omega-coinduct-var2)
    show \( x^\omega \leq (x \cdot x^*)^\omega \)
    by (metis mult-isol mult-oner omega-iso star-ref)
qed

lemma wagner-2-var: \( x \cdot (y \cdot x)^\omega \leq (x \cdot y)^\omega \)
proof —
    have \( x \cdot y \cdot x \leq x \cdot y \cdot x \)
    by auto
    thus \( x \cdot (y \cdot x)^\omega \leq (x \cdot y)^\omega \)
    by (metis mult.assoc omega-simulation)
qed
lemma wagner-2 [simp]: $x \cdot (y \cdot x) = (x \cdot y) \omega$
proof (rule antisym)
  show $x \cdot (y \cdot x) \omega \leq (x \cdot y) \omega$
    by (rule wagner-2-var)
  have $(x \cdot y) \omega = x \cdot y \cdot (x \cdot y) \omega$
    by (metis omega-unfold-eq)
  thus $(x \cdot y) \omega \leq x \cdot (y \cdot x) \omega$
    by (metis mult.assoc mult-isol wagner-2-var)
qed

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

lemma wagner-3:
assumes $x \cdot (x + y) \omega + z = (x + y) \omega$
shows $(x + y) \omega = x \cdot x + x^* \cdot z$
proof (rule antisym)
  show $(x + y) \omega \leq x \cdot x + x^* \cdot z$
    by (metis add.commute assms omega-coinduct-eq)
  have $x^* \cdot z \leq (x + y) \omega$
    by (metis add.commute assms star-inductl-eq)
  thus $x^* + x^* \cdot z \leq (x + y) \omega$
    by (metis add-lub omega-subdist)
qed

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

lemma wagner-1-var [simp]: $(x^* \cdot x) \omega \omega = x \omega$
proof (rule antisym)
  have $(x^* \cdot x) \omega \omega = 1 + x \omega \cdot (x^* \omega)\star$
    by simp
  also have $\ldots \leq 1 + x \omega \cdot \top$
    by (metis add-isos var eq-refl omega-1 omega-top)
  thus $(x^* \omega) \star \leq 1 + x \omega \cdot (x^* \omega)\star$
    by (metis calculation omega-top)
  show $1 + x \omega \leq (x^* \omega)\star$
    by (metis star2 star-ext)
qed

lemma star-omega-4 [simp]: $(x^\omega)^* = 1 + x^\omega$
proof (rule antisym)
  have $(x^\omega)^* = 1 + x^\omega \cdot (x^\omega)^*$
    by simp
  also have $\ldots \leq 1 + x^\omega \cdot \top$
    by (metis add-isos var eq-refl omega-1 omega-top)
  thus $(x^\omega)^* \leq 1 + x^\omega \cdot (x^\omega)^*$
    by (metis calculation omega-top)
  show $1 + x^\omega \leq (x^\omega)^*$
    by (metis star2 star-ext)
qed

lemma star-omega-5 [simp]: $x^\omega \cdot (x^\omega)^* = x^\omega$
proof (rule antisym)
  show $x^\omega \cdot (x^\omega)^* \leq x^\omega$
    by (rule omega-1)
  show $x^\omega \leq x^\omega \cdot (x^\omega)^*$
    by (metis mult.oner star-ref mult-isol)
qed

The next law shows how omegas below a sum can be unfolded.
lemma omega-sum-unfold: \( x^\omega + x^* \cdot y \cdot (x + y)^\omega = (x + y)^\omega \)

proof
- have \((x + y)^\omega = x \cdot (x + y)^\omega + y \cdot (x + y)^\omega\)
  by (metis distrib-right omega-unfold-eq)
thus thesis
  by (metis mult.assoc wagner-3)
qed

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

lemma omega-sum-unfold-coind: \((x + y)^\omega \leq (x^* \cdot y)^\omega + (x^* \cdot y)^* \cdot x^\omega\)
by (metis omega-coinduct-eq omega-sum-unfold)

lemma omega-sum-unfold-ind: \((x^* \cdot y)^* \cdot x^\omega \leq (x + y)^\omega\)
by (metis omega-sum-unfold-star-inductl-eq)

lemma wagner-1-gen: \((x \cdot y^*)^\omega \leq (x + y)^\omega\)
proof
- have \((x \cdot y^*)^\omega \leq ((x + y) \cdot (x + y)^*)^\omega\)
  by (metis add-ub1 add-ub2 mult-isol-var omega-iso star-iso)
thus thesis
  by (metis wagner-1)
qed

lemma wagner-1-var-gen: \((x^* \cdot y)^\omega \leq (x + y)^\omega\)
proof
- have \((x^* \cdot y)^\omega = x^* \cdot (y \cdot x^*)^\omega\)
  by (metis wagner-2)
also have ... \(\leq x^* \cdot (x + y)^\omega\)
  by (metis add.commute mult-isol wagner-1-gen)
also have ... \((x + y)^* \cdot (x + y)^\omega\)
  by (metis add-ub1 mult-isol star-iso)
thus thesis
  by (metis calculation order-trans star-omega-1)
qed

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

lemma omega-denest [simp]: \((x + y)^\omega = (x^* \cdot y)^\omega + (x^* \cdot y)^* \cdot x^\omega\)
proof (rule antisym)
  show \((x + y)^\omega \leq (x^* \cdot y)^\omega + (x^* \cdot y)^* \cdot x^\omega\)
    by (rule omega-sum-unfold-coind)
  have \((x^* \cdot y)^\omega \leq (x + y)^\omega\)
    by (rule wagner-1-var-gen)
hence \((x^* \cdot y)^* \cdot x^\omega \leq (x + y)^\omega\)
    by (metis omega-sum-unfold-ind)
thus \((x^* \cdot y)^\omega + (x^* \cdot y)^* \cdot x^\omega \leq (x + y)^\omega\)
    by (metis add-lub wagner-1-var-gen)
qed
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** \(y \cdot x \leq x \cdot (x + y)^*\)

**Shows** \((x + y)^\omega = x^\omega + x^* \cdot y^\omega\)

**Proof** (rule antisym)

- **Have** \(y^* \cdot x \leq x \cdot (x + y)^*\)
  - by (metis assms quasicomm-var)
- **Also have** \((x + y)^\omega = y^\omega + y^* \cdot x \cdot (x + y)^\omega\)
  - by (metis add.commute omega-sum-unfold)
- **Moreover have** \(\ldots \leq x \cdot (x + y)^* \cdot (x + y)^\omega + y^\omega\)
  - by (metis add-iso add-lub add-ub2 calculation(1) mult-isol)
- **Moreover have** \(\ldots \leq x \cdot (x + y)^\omega + y^\omega\)
  - by (metis mult.assoc order-refl star-omega-1)
- **Thus** \((x + y)^\omega \leq x^\omega + x^* \cdot y^\omega\)
  - by (metis add.commute calculation mult.assoc omega-coinduct star-omega-1)
- **Have** \(x^\omega \leq (x + y)^\omega\)
  - by (rule omega-subdist)
- **Moreover have** \(x^* \cdot y^\omega \leq x^* \cdot (x + y)^\omega\)
  - by (metis calculation add-ub1 mult-isol)
- **Moreover have** \(\ldots \leq (x + y)^* \cdot (x + y)^\omega\)
  - by (metis add-ub1 star-iso mult-isol)
- **Moreover have** \(\ldots = (x + y)^\omega\)
  - by (rule star-omega-1)
- **Thus** \(x^\omega + x^* \cdot y^\omega \leq (x + y)^\omega\)
  - by (metis add.commute add-lub calculation mult-isol omega-subdist order-trans star-omega-1)

**Qed**

The following theorem by Bachmair and Dershowitz [2] is a corollary.

**Lemma bachmair-dershowitz:**

**Assumes** \(y \cdot x \leq x \cdot (x + y)^*\)

**Shows** \((x + y)^\omega = 0 \iff x^\omega + y^\omega = 0\)

**Proof**

- **Assume** \((x + y)^\omega = 0\)
  - **Show** \(x^\omega + y^\omega = 0\)
    - by (metis \((x + y)^\omega = (0::'a)\) add.commute add-zero-r annir omega-sum-unfold)
- **Next**
  - **Assume** \(x^\omega + y^\omega = 0\)
    - **Show** \((x + y)^\omega = 0\)
      - by (metis \(x^\omega + y^\omega = (0::'a)\) assms no-trivial-inverse omega-sum-refine distrib-left star-omega-1)

**Qed**

The next lemmas consider an abstract variant of the empty word property from language theory and match it with the absence of infinite iteration [26].

**Definition** (in dioid-one-zero) exp
where \( \text{ewp} \ x \equiv \neg(\forall y. \ y \leq x \cdot y \rightarrow y = 0) \)

**lemma** \( \text{ewp-super-id1} \): \( 0 \neq 1 \rightarrow 1 \leq x \rightarrow \text{ewp} \ x \)

by (metis \( \text{ewp-def} \) mult-oner)

**lemma** \( 0 \neq 1 \rightarrow 1 \leq x \leftrightarrow \text{ewp} \ x \)

nitpick [expect=genuine] — 3-element counterexample

oops

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

**lemma** \( \text{ewp-neg-and-omega} \): \( \neg \text{ewp} \ x \leftrightarrow x^\omega = 0 \)

proof

assume \( \neg \text{ewp} \ x \)

hence \( \forall y. \ y \leq x \cdot y \rightarrow y = 0 \)

by (metis \( \text{ewp-def} \))

thus \( x^\omega = 0 \)

by (metis omega-fold)

next

assume \( x^\omega = 0 \)

hence \( \forall y. \ y \leq x \cdot y \rightarrow y = 0 \)

by (metis omega-coinduct-var zzero-unique)

thus \( \neg \text{ewp} \ x \)

by (metis \( \text{ewp-def} \))

qed

**lemma** \( \text{ewp-alt1} \): \( \forall z. \ x^\omega \leq x^* \cdot z \leftrightarrow (\forall y z. \ y \leq x \cdot y + z \rightarrow y \leq x^* \cdot z) \)

by (metis add-comm less-eq-def omega-coinduct omega-unfold-eq order-prop)

**lemma** \( \text{ewp-alt} \): \( x^\omega = 0 \leftrightarrow (\forall y z. \ y \leq x \cdot y + z \rightarrow y \leq x^* \cdot z) \)

by (metis annir antisym \( \text{ewp-alt1} \) zero-least)

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

**lemma** \( \text{omega-super-id1} \): \( 0 \neq 1 \rightarrow 1 \leq x \rightarrow x^\omega \neq 0 \)

by (metis eq-iff max-element omega-iso zero-least)

**lemma** \( \text{omega-super-id2} \): \( 0 \neq 1 \rightarrow x^\omega = 0 \rightarrow \neg(1 \leq x) \)

by (metis \( \text{omega-super-id1} \))

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

**lemma** \( \text{ardens-lemma-var} \):

assumes \( x^\omega = 0 \) and \( z + x \cdot y = y \)

shows \( x^* \cdot z = y \)

proof

have \( y \leq x^\omega + x^* \cdot z \)

by (metis assms omega-coinduct order-refl)

hence \( y \leq x^* \cdot z \)
by (metis add-zero-l assms)
thus \( x^* \cdot z = y \)
  by (metis assms eq_iff star_inductl_eq)
qed

lemma ardens-lemma: \( \neg \text{ewp } x \rightarrow z + x \cdot y = y \rightarrow x^* \cdot z = y \)
  by (metis ardens-lemma-var ewp_neg_and_omega)

lemma ardens-lemma-equiv:
  assumes \( \neg \text{ewp } x \)
  shows \( z + x \cdot y = y \leftrightarrow x^* \cdot z = y \)
proof
  assume \( z + x \cdot y = y \)
  thus \( x^* \cdot z = y \)
    by (metis ardens-lemma assms)
next
  assume \( x^* \cdot z = y \)
  also have \( z + x \cdot y = z + x \cdot x^* \cdot z \)
    by (metis calculation mult_assoc)
  moreover have \( ... = (1 + x \cdot x^*) \cdot z \)
    by (metis distrib_right mult_onel)
  moreover have \( ... = x^* \cdot z \)
    by (metis star_unfoldl_eq)
  thus \( z + x \cdot y = y \)
    by (metis calculation)
qed

lemma ardens-lemma-var-equiv: \( x^\omega = 0 \rightarrow (z + x \cdot y = y \leftrightarrow x^* \cdot z = y) \)
  by (metis ardens-lemma-var-equiv ewp_neg_and_omega)

lemma arden-conv1: (\( \forall y \ z. z + x \cdot y = y \rightarrow x^* \cdot z = y \)) \( \rightarrow \neg \text{ewp } x \)
  by (metis add-zero-l annir ewp_neg_and_omega omega_unfold_eq)

lemma arden-conv2: (\( \forall y \ z. z + x \cdot y = y \rightarrow x^* \cdot z = y \)) \( \rightarrow x^\omega = 0 \)
  by (metis arden-conv1 ewp_neg_and_omega)

lemma arden-var3: (\( \forall y \ z. z + x \cdot y = y \rightarrow x^* \cdot z = y \)) \( \leftrightarrow x^\omega = 0 \)
  by (metis arden-conv2 ardens-lemma-var)

end

10.2 Omega Algebras

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

end
11 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.

11.1 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 [18]. Thus, the omega of a relation is most naturally defined coinductively.

coinductive-set omega :: \( (\alpha \times \alpha) \set \Rightarrow (\alpha \times \alpha) \set \) for \( R \)

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

lemma omega-cases: \( (x, z) \in \omega R \Rightarrow (\forall y. (x, y) \in R \Rightarrow (y, z) \in \omega R \Rightarrow P) \Rightarrow P \)
by (metis omega.cases)

lemma omega-coinduct: \( X x z \Rightarrow (\forall x z. X x z \Rightarrow \exists y. (x, y) \in R \land (X y z \vee (y, z) \in \omega R)) \Rightarrow (x, z) \in \omega R \)
by (metis omega.coinduct)

lemma omega-weak-coinduct: \( X x z \Rightarrow (\forall x z. X x z \Rightarrow \exists y. (x, y) \in R \land X y z) \Rightarrow (x, z) \in \omega R \)
by (metis omega.coinduct)

lemma context-conjI-R:
assumes \( Q Q \Rightarrow P \)
shows \( P \)
by (iprover intro: conjI assms)

interpretation rel-omega-algebra: omega-algebra op \( \cup \) op \( O \) Id \{\} op \( \subseteq \) op \( \subset \)
rtrancl omega
proof
fix \( x y z :: \alpha \) rel
show \( \omega x \subseteq x \ O \omega x \)
by (auto elim: omega-cases)
show \( y \subseteq z \cup x \ O y \Rightarrow y \subseteq \omega x \cup x^* \ O z \)
apply auto
apply (rule omega-weak-coinduct[where \( X=\lambda a b. (a, b) \in x \ O y \land (a, b) \notin x^* \ O z \)]
apply (metis UnE in_mono relcompI rtrancl_refl)

88
apply \((\text{thin-tac } (a, b) \in y)\)
apply \((\text{thin-tac } (a, b) \notin x^* O z)\)
apply \text{clarsimp}
apply \text{rename-tac } a \ b \ e
apply \text{rule-tac } x=b \text{ in exI}
apply \text{simp}
apply \text{rule context-conjI-R}
apply \text{metis rel-dioid.mult.assoc relcompI rtrancl-into-rtrancl rtrancl-refl rtrancl-idemp-self-comp}
apply \text{metis UnE in-mono relcompI rtrancl-refl}
done
qed
done

end

12 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.

12.1 Auxiliary Lemmas

\textbf{lemma} \textit{fun-im}: \(\{f \ a \ |. \ a \in A\} = \{b. \ b \in f \cdot A\}\)
by \textit{auto}

\textbf{lemma} \textit{fset-to-im}: \(\{f x \ |. \ x \in X\} = f \cdot X\)
by \textit{auto}

\textbf{lemma} \textit{cart-flip-aux}: \(\{f \ (\text{snd } p) \ (\text{fst } p) \ |. \ p \in (B \times A)\} = \{f \ (\text{fst } p) \ (\text{snd } p) \ |. \ p \in (A \times B)\}\)
by \textit{auto}

\textbf{lemma} \textit{cart-flip}: \((\lambda p. f \ (\text{snd } p) \ (\text{fst } p)) \cdot (B \times A) = (\lambda p. f \ (\text{fst } p) \ (\text{snd } p)) \cdot (A \times B)\)
by \textit{(metis \textit{cart-flip-aux} fset-to-im)}

\textbf{lemma} \textit{fprod-aux}: \(\{x \cdot y \ |. \ x \cdot y \in (f \cdot A) \land y \in (g \cdot B)\} = \{f x \cdot g y \ |. \ x \cdot y \in A \land y \in B\}\)
by \textit{auto}
12.2 Finite Suprema in Semilattices

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

**lemma** setsum-sup:

assumes finite (A :: 'a::join-semilattice-zero set)
shows \( \sum A \leq z \leftrightarrow (\forall a \in A. a \leq z) \)

**proof** (induct rule: finite-induct[OF assms])

fix \( z \) :: 'a

show \( (\sum \{\} \leq z) = (\forall a \in \{\}. a \leq z) \)
by (metis setsum.empty zero-least Int-empty-right disjoint-iff-not-equal)

next

fix \( x z \) :: 'a and \( F \) :: 'a set

assume finF: finite \( F \)
and xnF: \( x \notin F \)
and indhyp: \( (\sum F \leq z) = (\forall a \in F. a \leq z) \)

show \( (\sum (insert x F) \leq z) = (\forall a \in insert x F. a \leq z) \)

**proof**

have \( \sum (insert x F) \leq z \leftrightarrow (x + \sum F) \leq z \)
by (metis finF setsum.insert xnF)

also have \( \ldots \leftrightarrow \sum F \leq z \)
by (metis add-lub)

also have \( \ldots \leftrightarrow (\forall a \in F. a \leq z) \)
by (metis (lifting) indhyp)

also have \( \ldots \leftrightarrow (\forall a \in insert x F. a \leq z) \)
by (metis insert-iff)

ultimately show \( (\sum (insert x F) \leq z) = (\forall a \in insert x F. a \leq z) \)
by blast

qed

This immediately implies some variants.

**lemma** setsum-less-eqI:

\((\forall x. x \in A \Rightarrow f x \leq y) \Rightarrow setsum f A \leq (y::'a::join-semilattice-zero)\)

**apply** (atomize (full))

**apply** (case-tac finite \( A \))

**apply** (erule finite-induct)

**apply** simp-all

**apply** (metis add-lub)

**done**

**lemma** setsum-less-eqE:

\[ \text{setsum} f A \leq y; x \in A; \text{finite} A \] \( \Rightarrow \) \( f x \leq (y::'a::join-semilattice-zero) \)

**apply** (erule rev-mp)

**apply** (erule rev-mp)

**apply** (erule finite-induct)

**apply** (auto simp add: add-lub)

**done**
lemma setsum-fun-image-sup:
fixes f :: 'a ⇒ 'b :: join-semilattice-zero
assumes finite (A :: 'a set)
shows \( \sum (f ' A) \leq z \leftrightarrow (\forall a \in A. f a \leq z) \)
by (simp add: assms setsum-sup)

lemma setsum-fun-sup:
fixes f :: 'a ⇒ 'b :: join-semilattice-zero
assumes finite (A :: 'a set)
shows \( \sum \{ f a | a \in A \} \leq z \leftrightarrow (\forall a \in A. f a \leq z) \)
by (simp only: fset-to-im assms setsum-fun-image-sup)

lemma setsum-intro:
assumes finite (A :: 'a :: join-semilattice-zero set) and finite B
shows \( (\forall a \in A. \exists b \in B. a \leq b) \rightarrow (\sum A \leq \sum B) \)
by (metis assms order-refl order-trans setsum-sup)

Next we prove an additivity property for suprema.

lemma setsum-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 –
  have \( \forall z. \sum (A \cup B) \leq z \leftrightarrow (\sum A + \sum B \leq z) \)
  by (auto simp add: assms setsum-sup add-lub)
thus \( \text{thesis} \)
  by (simp add: eq-iff)
qed

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

lemma setsum-bin[simp]: \( \sum \{ (x :: 'a::join-semilattice-zero), y \} = x + y \)
by (subst insert-is-Un, subst setsum-union, auto)

Next we show that finite suprema are order preserving.

lemma setsum-iso:
assumes finite (B :: 'a::join-semilattice-zero set)
show \( A \subseteq B \rightarrow (\sum A \leq \sum B) \)
by (metis assms finite-subset order-refl order-trans setsum-sup)

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 setsum-insert [simp]:
assumes finite (A :: 'a::join-semilattice-zero set)
shows \( \sum (\text{insert } x \ A) = x + \sum A \)
proof –
  have \( \sum (\text{insert } x \ A) = \sum \{ x \} + \sum A \)

by (metis insert-is-Un assms finite.emptyI finite.insertI setsum-union)
thus ?thesis
by auto
qed

lemma setsum-fun-insert:
  fixes f :: 'a ⇒ 'b::join-semilattice-zero
  assumes finite (A :: 'a set)
  shows \( \sum (f \cdot (\text{insert } x A)) = f x + \sum (f \cdot A) \)
  by (simp add: assms)

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 \( \sum ((\lambda x. \sum (f x \cdot B)) \cdot A) = \sum ((\lambda p. f \cdot \text{fst} p \cdot \text{snd} p) \cdot (A \times B)) \)
proof
  have \( \forall z. \sum ((\lambda x. \sum (f x \cdot B)) \cdot A) \leq z \iff \sum ((\lambda p. f \cdot \text{fst} p \cdot \text{snd} p) \cdot (A \times B)) \leq z \)
    by (simp add: assms finite-cartesian-product setsum-fun-image-sup)
  thus ?thesis
    by (simp add: eq-iff)
qed

lemma flatten2-im:
  fixes f :: 'a ⇒ 'a ⇒ 'b::join-semilattice-zero
  assumes finite A
  and finite B
  shows \( \sum \{ \sum \{ f x y \mid y. y \in B \mid x. x \in A \} \mid x. x \in A \land y \in B \} \)
  apply (simp only: flatten1-im assms cart-flip)
  done

lemma setsum-flatten1:
  fixes f :: 'a ⇒ 'a ⇒ 'b::join-semilattice-zero
  assumes finite (A :: 'a set)
  and finite (B :: 'a set)
  shows \( \sum \{ \sum \{ f x y \mid y. y \in B \mid x. x \in A \} \mid x. x \in A \land y \in B \} \)
  apply (simp add: fset-to-im assms flatten1-im)
  apply (subst fset-to-im[symmetric])
  apply simp
  done

lemma setsum-flatten2:
  fixes f :: 'a ⇒ 'a ⇒ 'b::join-semilattice-zero
  assumes finite A
  and finite B
  shows \( \sum \{ \sum \{ f x y \mid x. x \in A \} \mid y. y \in B \} \)
  apply (simp add: fset-to-im assms flatten2-im)
  done

92
apply \((\text{subst } f\text{-set-to-im}[\text{symmetric}])\)
apply simp
done

Next we show another additivity property for suprema.

**lemma** setsum-fun-sum:

fixes \(f, g : \mathcal{A} \Rightarrow \mathcal{B}\) :: \(\text{join-semilattice-zero}\)
assumes finite \((A : \mathcal{A} \text{ set})\)
sows \(\sum ((\lambda x. f x + g x) \cdot A) = \sum (f \cdot A) + \sum (g \cdot A)\)

proof –

\(
\begin{align*}
\{ \\
\ & \text{fix } z :: \mathcal{B} \\
\ & \text{have } \sum ((\lambda x. f x + g x) \cdot A) \leq z \leftrightarrow \sum (f \cdot A) + \sum (g \cdot A) \leq z \\
\ & \text{by } (\text{auto simp add: assms setsum-fun-image-sup add-lub}) \\
\}
\end{align*}
\)

thus ?thesis
by (simp add: eq-iff)
qed

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** setsum-fun-add:

fixes \(f :: \mathcal{A} \Rightarrow \mathcal{B}\) :: \(\text{join-semilattice-zero}\)
assumes finite \((X : \mathcal{A} \text{ set})\)
and fstrict: \(f \ 0 = 0\)
and fadd: \(\forall x y. f (x + y) = f x + f y\)
sows \(f (\sum X) = \sum (f \cdot X)\)

proof (induct rule: finite-induct[OF assms(1)])

show \(f (\sum \{\}) = \sum (f \cdot \{\})\)
by (metis fstrict image-empty setsum.empty)

fix \(x :: \mathcal{A}\) and \(F :: \mathcal{A} \text{ set}\)
assume finF: finite \(F\)
and indhyp: \(f (\sum F) = \sum (f \cdot F)\)

have \(f (\sum (\text{insert } x F)) = f (x + \sum F)\)
by (metis setsum-insert finF)
also have \(\ldots = f x + (f (\sum F))\)
by (rule fadd)
also have \(\ldots = f x + \sum (f \cdot F)\)
by (metis indhyp)
also have \(\ldots = \sum (f \cdot (\text{insert } x F))\)
by (metis finF setsum-fun-insert)

finally show \(f (\sum (\text{insert } x F)) = \sum (f \cdot \text{insert } x F)\).
qed

12.3 Finite Suprema in Dioids

In this section we mainly prove variants of distributivity laws.
lemma setsum-distl:
assumes finite Y
shows \((x :: 'a::dioid-one-zero) \cdot (\sum Y) = \sum \{x \cdot y | y \in Y\}\)
by (simp only: setsum-fun-add assms annil distrib-left Collect-mem-eq fun-im)

lemma setsum-distr:
assumes finite X
shows \((\sum X) \cdot (y :: 'a::dioid-one-zero) = \sum \{x \cdot y | x \in X\}\)
proof –
have \((\sum X) \cdot y = \sum ((\lambda x. x \cdot y) \cdot X)\)
  by (rule setsum-fun-add,metis assms)
thus \(?thesis\)
  by (metis Collect-mem-eq fun-im)
qed

lemma setsum-fun-distl:
fixes f :: 'a => 'b::dioid-one-zero
assumes finite \((Y :: 'a set)\)
shows \(x \cdot \sum (f \cdot Y) = \sum \{x \cdot f y | y \in Y\}\)
by (simp add: assms fun-im image-image setsum-distl)

lemma setsum-fun-distr:
fixes f :: 'a => 'b::dioid-one-zero
assumes finite \((X :: 'a set)\)
shows \(\sum (f \cdot X) \cdot y = \sum \{f x \cdot y | x \in X\}\)
by (simp add: assms fun-im image-image setsum-distr)

lemma setsum-distl-flat:
assumes finite \((X :: 'a::dioid-one-zero set)\)
  and finite Y
shows \(\sum \{x \cdot \sum Y | x \in X\} = \sum \{x \cdot y | y \in Y\}\)
by (simp only: assms setsum-distl setsum-flatten1)

lemma setsum-distr-flat:
assumes finite X
  and finite \((Y :: 'a::dioid-one-zero set)\)
shows \(\sum \{\sum X \cdot y | y \in Y\} = \sum \{x \cdot y | x \in X \land y \in Y\}\)
by (simp only: assms setsum-distr setsum-flatten2)

lemma setsum-sum-distl:
assumes finite \((X :: 'a::dioid-one-zero set)\)
  and finite Y
shows \(\sum ((\lambda x. x \cdot (\sum Y)) \cdot X) = \sum \{x \cdot y | y \in Y\}\)
proof –
have \(\sum ((\lambda x. x \cdot (\sum Y)) \cdot X) = \sum \{\sum \{x \cdot y | y \in Y\} | x \in X\}\)
  by (auto simp add: setsum-distl assms fset-to-im)
thus \(?thesis\)
  by (simp add: assms setsum-flatten1)
qed
lemma setsum-sum-distr:
  assumes finite X
  and finite Y
  shows \( \sum \left( \lambda y. (\sum X) \cdot (y :: 'a::dioid-one-zero) \right) \cdot Y \) = \( \sum \{ x \cdot y \mid x \in X \land y \in Y \} \)
proof
  have \( \sum \left( \lambda y. (\sum X) \cdot (y :: 'a::dioid-one-zero) \right) \cdot Y \) = \( \sum \{ \sum \{ x \cdot y \mid x \in X \} \cdot y \cdot Y \mid y \cdot y \in Y \} \)
  by (auto simp add: setsum-distr assms fset-to-im)
  thus \( ?thesis \)
  by (simp add: assms setsum-flatten2)
qed

lemma setsum-sum-distl-fun:
  fixes f g :: 'a ⇒ 'b::dioid-one-zero
  fixes h :: 'a ⇒ 'a set
  assumes \( \forall x. \text{finite } (h x) \)
  and finite X
  shows \( \sum \left( \lambda x. f x \cdot \sum (g ' (h x)) \right) \cdot X \) = \( \sum \{ \sum \{ f x \cdot g y \mid x \cdot x \in A \land y \cdot y \in B \} \mid x \cdot x \in A \} \)
  by (auto simp add: setsum-fun-distl assms fset-to-im)

lemma setsum-sum-distr-fun:
  fixes f g :: 'a ⇒ 'b::dioid-one-zero
  fixes h :: 'a ⇒ 'a set
  assumes finite Y
  and \( \forall y. \text{finite } (h y) \)
  shows \( \sum \left( \lambda y. \sum (f ' (h y)) \cdot g y \right) \cdot Y \) = \( \sum \{ \sum \{ f x \cdot g y \mid x \cdot x \in (h y) \} \cdot y \cdot y \in Y \} \)
  by (auto simp add: setsum-fun-distr assms fset-to-im)

lemma setsum-dist:
  assumes finite (A :: 'a::dioid-one-zero set)
  and finite B
  shows \( \sum A \cdot (\sum B) = \sum \{ x \cdot y \mid x \cdot x \in A \land y \cdot y \in B \} \)
proof
  have \( \sum A \cdot (\sum B) = \sum \{ x \cdot \sum B \mid x \cdot x \in A \} \)
  by (simp add: assms setsum-distr)
  also have \( \ldots = \sum \{ \sum \{ x \cdot y \mid y \cdot y \in B \} \mid x \cdot x \in A \} \)
  by (simp add: assms setsum-distl)
  finally show \( ?thesis \)
  by (simp only: setsum-flatten1 assms finite-cartesian-product)
qed

lemma dioid-setsum-prod-var:
  fixes f g :: 'a ⇒ 'b::dioid-one-zero
  assumes finite (A :: 'a set)
  shows \( \sum (f ' A) \cdot (\sum (g ' A)) = \sum \{ f x \cdot g y \mid x \cdot x \in A \land y \cdot y \in A \} \)
  by (simp add: assms setsum-dist fprod-aux)
Lemma \texttt{dioid-setsum-prod}:

\begin{itemize}
\item \textbf{fixes} $f, g :: 'a \Rightarrow 'b$:
\item \textbf{assumes} finite ($A :: 'a set$)
\item \textbf{shows} ($\big{\sum} \{f x \mid x, y \in A\} \cdot \big{(\sum} \{g x \mid x, y \in A\} = \sum \{f x \cdot g y \mid x, y \in A\}$
\end{itemize}

by (simp add: assms dioid-setsum-prod-var fset-to-im)

There are interesting theorems for finite sums in Kleene algebras; we leave them for future consideration.

end

13 \ Formal Power Series

theory \textit{Formal-Power-Series}

imports \textit{Finite-Suprema Kleene-Algebra}

begin

13.1 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 \textit{HOL/Library/Formal_Power_Series.thy}.

\textbf{typedef} ($'a, 'b) fps = f::'a list \Rightarrow 'b. True$

\textbf{morphisms} fps-nth Abs-fps

by simp

It is often convenient to reason about functions, and transfer results to formal power series.

\textbf{setup-lifting} \textit{type-definition-fps}

\textbf{declare} fps-nth-inverse [simp]

\textbf{notation} fps-nth (infixl \$ 75)

\textbf{lemma} expand-fps-eq: $p = q \iff (\forall n. p \$ n = q \$ n)$

by (simp add: fps-nth-inject [symmetric] fun-eq-iff)

\textbf{lemma} fps-ext: $(\forall n. p \$ n = q \$ n) \Rightarrow p = q$

by (simp add: expand-fps-eq)

\textbf{lemma} fps-nth-Abs-fps [simp]: Abs-fps $f \$ n = f n

by (simp add: Abs-fps-inverse)

96
13.2 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.

```plaintext
instantiation fps :: (type,zero) zero
begin
  definition zero-fps where
    0 = Abs-fps (λn. 0)
  instance ..
end
```

```plaintext
lemma fps-zero-nth [simp]: 0 $ n = 0
unfolding zero-fps-def by simp
```

The unit formal power series maps the monoidal unit (the empty list) to one and all other elements to zero.

```plaintext
instantiation fps :: (type, {one,zero}) one
begin
  definition one-fps where
    1 = Abs-fps (λn. if n = [] then 1 else 0)
  instance ..
end
```

```plaintext
lemma fps-one-nth-Nil [simp]: 1 $ [] = 1
unfolding one-fps-def by simp
```

```plaintext
lemma fps-one-nth-Cons [simp]: 1 $ (x # xs) = 0
unfolding one-fps-def by simp
```

Addition of formal power series is the usual pointwise addition of functions.

```plaintext
instantiation fps :: (type,plus) plus
begin
  definition plus-fps where
    f + g = Abs-fps (λn. f $ n + g $ n)
  instance ..
end
```

```plaintext
lemma fps-add-nth [simp]: (f + g) $ n = f $ n + g $ n
unfolding plus-fps-def by simp
```

This directly shows that formal power series form a semilattice with zero.

```plaintext
lemma fps-add-assoc: ((f::('a,'b::semigroup-add) fps) + g) + h = f + (g + h)
unfolding plus-fps-def by (simp add: add.assoc)
```

```plaintext
lemma fps-add-comm [simp]: (f::('a,'b::ab-semigroup-add) fps) + g = g + f
unfolding plus-fps-def by (simp add: add.commute)
```
lemma fps-add-idem [simp]: \( (\cdot ('a', 'b::{join-semilattice}) \text{fps}) + f = f \)
unfolding plus-fps-def by simp

lemma fps-zeroL [simp]: \( (\cdot ('a', 'b::{monoid-add}) \text{fps}) + 0 = f \)
unfolding plus-fps-def by simp

lemma fps-zeroR [simp]: \( 0 + (\cdot ('a', 'b::{monoid-add}) \text{fps}) = f \)
unfolding plus-fps-def by simp

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
  \( f * g = \text{Abs-fps} (\lambda n. \sum \{ f \cdot y * g \cdot z \mid y @ z \cdot n = y @ z \}) \)
  instance ..
end

We call the set of all prefix/suffix splittings of a list \( xs \) the \( \text{splitset} \) of \( xs \).

definition splitset where
\( \text{splitset} \; xs \equiv \{ (p, q). \; \text{xs} = p @ q \} \)

Alternatively, splitsets can be defined recursively, which yields convenient simplification rules in Isabelle.

fun splitset-fun where
\( \text{splitset-fun} \; [] = \{ ([], []) \} \)
| \( \text{splitset-fun} \; (x \# xs) = \text{insert} \; ([], x \# xs) \; (\text{apfst} \; (\text{Cons} \; x) \; \text{splitset-fun} \; xs) \)

lemma splitset-consl:
\( \text{splitset} \; (x \# xs) = \text{insert} \; ([], x \# xs) \; (\text{apfst} \; (\text{Cons} \; x) \; \text{splitset} \; xs) \)
by (auto simp add: image-def splitset-def (metis append-eq-Cons-conv)+

lemma splitset-eq-splitset-fun: \( \text{splitset} \; xs = \text{splitset-fun} \; xs \)
apply (induct xs)
apply (simp add: splitset-def)
apply (simp add: splitset-consl)
done

The definition of multiplication is now more precise.

lemma fps-mult-var:
\( (f * g) \cdot n = \sum \{ f \cdot (\text{fst} \; p) * g \cdot (\text{snd} \; p) \mid p. \; p \in \text{splitset} \; n \} \)
by (simp add: times-fps-def splitset-def)

lemma fps-mult-image:
\( (f * g) \cdot n = \sum ((\lambda p. f \cdot (\text{fst} \; p) * g \cdot (\text{snd} \; p)) \cdot \text{splitset} \; n) \)
by (simp only: Collect-mem-eq fps-mult-var fun-im)

Next we show that splitsets are finite and non-empty.

**lemma** splitset-fun-finite [simp]: finite (splitset-fun xs)
by (induct xs, simp-all)

**lemma** splitset-finite [simp]: finite (splitset xs)
by (simp add: splitset-eq-splitset-fun)

**lemma** split-append-finite [simp]: finite {p, q}. xs = p @ q
by (fold splitset-def, fact splitset-finite)

**lemma** splitset-fun-nonempty [simp]: splitset-fun xs ≠ {}
by (cases xs, simp-all)

**lemma** splitset-nonempty [simp]: splitset xs ≠ {}
by (simp add: splitset-eq-splitset-fun)

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
by (rule fps-ext) (simp add: times-fps-def setsum.neutral)

**lemma** fps-annir [simp]:
(f::('a::type,'b::{comm-monoid-add,mult-zero}) fps) * 0 = 0
by (simp add: fps-ext times-fps-def setsum.neutral)

**lemma** fps-distl:
(f::('a::type,'b::{join-semilattice-zero,semiring}) fps) * (g + h) = (f * g) + (f * h)
by (simp add: fps-ext fps-mult-image distrib-left setsum-fun-sum)

**lemma** fps-distr:
((f::('a::type,'b::{join-semilattice-zero,semiring}) fps) + g) * h = (f * h) + (g * h)
by (simp add: fps-ext fps-mult-image distrib-right setsum-fun-sum)

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** (rule fps-ext)
fix $n :: 'a list$

show $(1 * f) \% n = f \% n$

proof (cases $n$)
  case Nil thus ?thesis
  by (simp add: times-fps-def)

next
  case Cons thus ?thesis
  by (simp add: fps-mult-image splitset-eq-splitset-fun image-comp one-fps-def comp-def image-constant-conv)

qed

lemma fps-oner [simp]:
$(f :: ('a::type,'b::{join-semilattice-zero,monoid-mult,mult-zero}) fps) * 1 = f$

proof (rule fps-ext)
  fix $n :: 'a list$
  {
    fix $z :: 'b$
    have $(f * 1) \% n \leq z \iff (\forall p \in \text{splitset } n. f \% (fst p) * 1 \% (snd p) \leq z)$
    by (simp add: fps-mult-image setsum-fun-image-sup)
    also have ... \iff $(\forall a b. n = a @ b \rightarrow f \% a * 1 \% b \leq z)$
    unfolding splitset-def by simp
    also have ... \iff $(f \% n * 1 \% [] \leq z)$
    by (metis append-Nil2 fps-one-nth-Cons fps-one-nth-Nil mult-zero-right neq-Nil-conv zero-least)
  }
  finally have $(f * 1) \% n \leq z \iff f \% n \leq z$
  by simp

  thus $(f * 1) \% n = f \% n$
  by (metis eq-iff)

qed

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 $F :: 'a list \Rightarrow 'a list \Rightarrow 'a list \Rightarrow 'b::join-semilattice-zero$
shows $\sum (\{ \sum \{ F (fst p) (fst q) (snd q) \mid q \in \text{splitset} (snd p) \} \mid p. p \in \text{splitset} x \}) =$
$\sum (\{ \sum \{ F (fst p) (snd q) (snd p) \mid q. q \in \text{splitset} (fst p) \} \mid p. p \in \text{splitset} x \})$
(is ?lhs = ?rhs)

proof -
  {
    fix $z :: 'b$
    have $?lhs \leq z \iff (\forall p q r. x = p @ q @ r \rightarrow F p q r \leq z)$
    by (simp only: fset-to-im setsum-fun-image-sup splitset-finite)
    (auto simp add: splitset-def)
    hence $?lhs \leq z \iff ?rhs \leq z$
  }
by (simp only: fset-to-im setsum-fun-image-sup splitset-finite)
(auto simp add: splitset-def)
}
thus thesis
by (simp add: eq-iff)
qed

lemma fps-mult-assoc: (f::('a::type,'b::dioid-one-zero) fps) * (g * h) = (f * g) * h
proof (rule fps-ext)
fix n :: 'a list
have (f * (g * h)) $ n = ∑ (f $ (fst p) * g $ (fst q) * h $ (snd q) | q. q ∈ splitset (snd p)) | p. p ∈ splitset n
by (simp add: fps-mult-image setsum-sum-distl-fun mult.assoc)
also have ... = ∑ (f $ (fst q) * g $ (snd q) * h $ (snd p) | q. q ∈ splitset (fst p)) | p. p ∈ splitset n
by (fact splitset-rearrange)
finally show (f * (g * h)) $ n = ((f * g) * h) $ n
by (simp add: fps-mult-image setsum-sum-distr-fun mult.assoc)
qed

13.3 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
fix x :: 'a
show 0 * x = 0
by (fact annil)
show x * 0 = 0
by (fact annir)
qed

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
fix f g h :: ('a,'b) fps
show $f + g + h = f + (g + h)$
  by (fact fps-add-assoc)
show $f + g = g + f$
  by (fact fps-add-comm)
show $f * g * h = f * (g * h)$
  by (metis fps-mult-assoc)
show $(f + g) * h = f * h + g * h$
  by (fact fps-distr)
show $1 * f = f$
  by (fact fps-onel)
show $f * 1 = f$
  by (fact fps-oner)
show $0 + f = f$
  by (fact fps-zeror)
show $0 * f = 0$
  by (fact fps-annil)
show $f * 0 = 0$
  by (fact fps-annir)
show $f \leq g \iff f + g = g$
  by (fact less-eq-fps-def)
show $f < g \iff f \leq g \land f \neq g$
  by (fact less-fps-def)
show $f + f = f$
  by (fact fps-add-idem)
show $f * (g + h) = f \cdot g + f \cdot h$
  by (fact fps-distl)
qed

end

lemma expand-fps-less-eq: $(f :: ('a, 'b :: dioid-one-zero) fps) \leq g \iff (\forall n. f \# n \leq g \# n)$
by (simp add: expand-fps-eq less-eq-def less-eq-fps-def)

13.4 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 Setsum-splitlist-nonempty:
\[
\sum \{ f \# ys \# zs \mid ys \# zs, zs = ys \# \emptyset \# zs \} = ((f \# xs) :: 'a :: join-semilattice-zero) + \sum \{ f \# ys \# zs \mid ys \# zs, zs = ys \# \emptyset \# zs \land ys \neq \emptyset \}
\]
proof
  have \{
    \{ f \# ys \# zs \mid ys \# zs, zs = ys \# \emptyset \# zs \} = \{ f \# ys \# zs \mid ys \# zs, zs = ys \# \emptyset \# zs \land ys = \emptyset \} \}
  by blast
thus thesis using [[simproc add: finite-Collect]]
   by (simp add: setsum.insert)
qed

lemma (in left-kleene-algebra) add-star-eq:
  \( x + y \cdot y^* \cdot x = y^* \cdot x \)
by (metis add.commute mult-onel star2 star-one troeger)

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.

declarerev-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 \( f \cdot [] \) = \( f \cdot [] \)^*  
| star-fps-rep-Cons: star-fps-rep \( f \cdot n \) = \( f \cdot [] \)^* \cdot \( \sum \{ f \cdot y \cdot \text{star-fps-rep} f \cdot z \cdot y \cdot z. \cdot n = y @ z \land y \neq [] \} \)

lift-definition star-fps :: ('a, 'b) fps \Rightarrow ('a, 'b) fps is star-fps-rep ..

lemma star-fps-Nil [simp]: \( f \cdot [] \) = \( f \cdot [] \)^*
by (simp add: star-fps-def)

lemma star-fps-Cons [simp]: \( f^* \cdot (x \# xs) \) = \( f^* \cdot (f \cdot [])^* \cdot \sum \{ f \cdot y \cdot f^* \cdot z \cdot y \cdot z. \cdot x \# xs = y @ z \land y \neq [] \} \)
by (simp add: star-fps-def)

instance
proof
  fix \( f \cdot g \cdot h :: ('a, 'b) fps \)
  have I + f \cdot f^* = f^*
    apply (rule fps-ext)
    apply (case-tac n)
    apply (auto simp add: times-fps-def)
    apply (simp add: add-star-eq mult.assoc[THEN sym] Setsum-splitlist-nonempty)
  done
thus I + f \cdot f^* \leq f^*
by (metis order-refl)

have \( f \cdot g \leq g \rightarrow f^* \cdot g \leq g \)
proof
  assume f \cdot g \leq g
  hence I: \( \forall u \cdot v. \cdot f \cdot g \cdot v \leq g \cdot (u @ v) \)
  using [[simproc add: finite-Collect]]
apply (simp add: expand-fps-less-eq)
apply (drule-tac x=u @ v in spec)
apply (simp add: times-fps-def)
apply (auto elim!: setsum-less-eqE)
done

hence 2: \( \forall v. (f \#[]) * g \# v \leq g \# v \)
apply (subgoal-tac f \# \cdot g \# v \leq g \# v)
apply (metis star-inductl-var)
apply (metis append-Nil)
done

show \( f \cdot g \leq g \)
using [[simproc add: finite-Collect]]
apply (auto intro!: setsum-less-eqI simp add: expand-fps-less-eq times-fps-def)
apply (induct-tac y rule: length-induct)
apply (case-tac xs)
apply (simp add: 2)
apply (auto simp add: mult.assoc setsum-distr)
apply (rule-tac y = (f \#[]) \cdot g \# (a # list @ z) in order-trans)
prefer 2
apply (rule 2)
apply (auto intro!: mult-isol[rule-format] setsum-less-eqI)
apply (drule-tac z = f \# y in spec)
apply (drule mp)
apply (metis 1 append-Cons append-assoc)
done

qed

thus \( h + f \cdot g \leq g \rightarrow f^* \cdot h \leq g \)
by (metis (hide-lams, no-types) add-lub mult-isol order-trans)

have \( g \cdot f \leq g \rightarrow g \cdot f^* \leq g \)
— this property is dual to the previous one; the proof is slightly different

proof
assume \( g \cdot f \leq g \)

hence 1: \( \forall u v. g \$ u \cdot f \$ v \leq g \$ (u @ v) \)
using [[simproc add: finite-Collect]]
apply (simp add: expand-fps-less-eq)
apply (drule-tac x=u @ v in spec)
apply (simp add: times-fps-def)
apply (auto elim!: setsum-less-eqE)
done

hence 2: \( \forall u. g \$ u \cdot (f \#[])^* \leq g \$ u \)
apply (subgoal-tac g \$ u \cdot f \# [] \leq g \$ u)
apply (metis star-inductr-var)
apply (metis append-Nil2)
done

show \( g \cdot f^* \leq g \)
using [[simproc add: finite-Collect]]
apply (auto intro!: setsum-less-eqI simp add: expand-fps-less-eq times-fps-def)
apply (rule-tac P=λy. g $ y · f $ z ≤ g $ (y ⊕ z) and x=y in allE)
prefer 2
apply assumption
apply (induct-tac z rule: length-induct)
apply (case-tac xs)
apply (simp add: 2)
apply (auto intro! setsum-less-eqI simp add: setsum-distl)
apply (rule-tac y=g $ x · f $ yb · f $ z in order-trans)
apply (simp add: 2 mult.assoc[THEN sym] mult-isor)
apply (rule-tac y=g $ (x ⊕ yb) · f $ z in order-trans)
apply (simp add: 1 mult-isor)
apply (drule-tac x=z in spec)
apply (drule mp)
apply (metis append-eq-Cons-conv length-append less-not-refl2 add.commute not-less-eq trans-less-add1)
apply (metis append-assoc)
done
qed
thus h + g · f ≤ g → h · f* ≤ g
by (metis (hide-lams, no-types) add-lub mult-isor order-trans)
qed
end
end

14 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 proof are similar to those for formal power series, but simpler.

class finite =
  assumes finiteuniv: finite UNIV

type-synonym ('a, 'b, 'c) matrix = 'a ⇒ 'b ⇒ 'c

definition mat-one :: ('a, 'a, 'c::dioid-one-zero) matrix (ε) where
\[ \epsilon_{ij} \equiv (\text{if } (i = j) \text{ then } 1 \text{ else } 0) \]

**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
\[ (f \oplus g) i j \equiv \lambda j. (f i j) + (g i j) \]

**Lemma** mat-add-assoc: \((f \oplus g) \oplus h = f \oplus (g \oplus h)\)
by (auto simp add: mat-add-def)

**Lemma** mat-add-comm: \(f \oplus g = g \oplus f\)
by (auto simp add: mat-add-def)

**Lemma** mat-add-idem[simp]: \(f \oplus f = f\)
by (auto simp add: mat-add-def)

**Lemma** mat-zerol[simp]: \(f \oplus \delta = f\)
by (auto simp add: mat-add-def mat-zero-def)

**Lemma** mat-zerror[simp]: \(\delta \oplus f = f\)
by (auto simp add: mat-add-def mat-zero-def)

**Definition** mat-mult :: \('a, 'k::finite, 'c::dioid-one-zero\) matrix \Rightarrow ('a, 'b, 'c) matrix \Rightarrow ('a, 'b, 'c) matrix (infixl \otimes 60) where
\[ (f \otimes g) i j \equiv \sum \{(f i k) \cdot (g k j) | k. k \in \text{UNIV}\} \]

**Lemma** mat-annil[simp]: \(\delta \otimes f = \delta\)
by (rule ext, auto simp add: mat-mult-def mat-zero-def)

**Lemma** mat-annir[simp]: \(f \otimes \delta = \delta\)
by (rule ext, auto simp add: mat-mult-def mat-zero-def)

**Lemma** mat-distl: \(f \otimes (g \oplus h) = (f \otimes g) \oplus (f \otimes h)\)
**Proof** –
\{
  fix \(i j\)
  have \((f \otimes (g \oplus h)) i j = \sum \{f i k \cdot (g k j + h k j) | k. k \in \text{UNIV}\}\)
    by (simp only: mat-mult-def mat-add-def)
  also have \(\cdots = \sum \{f i k \cdot g k j + f i k \cdot h k j | k. k \in \text{UNIV}\}\)
    by (simp only: distrib-left)
  also have \(\cdots = \sum \{f i k \cdot g k j | k. k \in \text{UNIV}\} + \sum \{f i k \cdot h k j | k. k \in \text{UNIV}\}\)
    by (simp only: fset-to-im setsum-fun-sum finiteuniv)
  finally have \((f \otimes (g \oplus h)) i j = ((f \otimes g) \oplus (f \otimes h)) i j\)
    by (simp only: mat-mult-def mat-add-def)
\}
thus ?thesis
by auto
qed

lemma mat-distr: \((f \oplus g) \otimes h = (f \otimes h) \oplus (g \otimes h)\)
proof -
{  
  fix \(i\) \(j\)
  have \(((f \oplus g) \otimes h) i j = \sum \{(f i k + g i k) \cdot h k j \mid k.\ k \in UNIV\}\)
    by (simp only: mat-mult-def mat-add-def)
  also have \(\ldots = \sum \{f i k \cdot h k j + g i k \cdot h k j \mid k.\ k \in UNIV\}\)
    by (simp only: distrib-right)
  also have \(\ldots = \sum \{f i k \cdot h k j \mid k.\ k \in UNIV\} + \sum \{g i k \cdot h k j \mid k.\ k \in UNIV\}\)
    by (simp only: fset-to-im setsum-fun-sum finiteuniv)
  finally have \(((f \oplus g) \otimes h) i j = ((f \otimes h) \oplus (g \otimes h)) i j\)
    by (simp only: mat-mult-def mat-add-def)
}\nthus \(?thesis\)
by auto
qed

lemma logic-aux1: \((\exists k. (i = k \rightarrow x = f i j) \land (i \neq k \rightarrow x = 0)) \leftrightarrow (\exists k. i = k \land x = f i j) \lor (\exists k. i \neq k \land x = 0)\)
by blast

lemma logic-aux2: \((\exists k. (k = j \rightarrow x = f i j) \land (k \neq j \rightarrow x = 0)) \leftrightarrow (\exists k. k = j \land x = f i j) \lor (\exists k. k \neq j \land x = 0)\)
by blast

lemma mat-onel[simp]: \(\varepsilon \otimes f = f\)
proof -
{  
  fix \(i\) \(j\)
  have \((\varepsilon \otimes f) i j = \sum \{x. (\exists k. (i = k \rightarrow x = f i j) \land (i \neq k \rightarrow x = 0))\}\}
    by (auto simp add: mat-mult-def mat-one-def)
  also have \(\ldots = \sum \{x. \exists k. (i = k \land x = f i j)\} \cup \{x. \exists k. (i \neq k \land x = 0)\}\)
    by (simp only: Collect-disj-eq logic-aux1)
  also have \(\ldots = \sum \{x. \exists k. (i = k \land x = f i j)\} + \sum \{x. \exists k. (i \neq k \land x = 0)\}\)
    by (rule setsun-union, auto)
  finally have \((\varepsilon \otimes f) i j = f i j\)
    by (auto simp add: setsum.neutral)
}\nthus \(?thesis\)
by auto
qed

lemma mat-oner[simp]: \(f \otimes \varepsilon = f\)
proof -
{  

fix $i\ j$

have $(f \otimes g) \ i\ j = \sum \{ x. (\exists k. (k = j \rightarrow x = f\ i\ j) \land (k \neq j \rightarrow x = 0))\}$
  by (auto simp add: mat-mult-def mat-one-def)
also have \dots = $\sum \{(x. \exists k. (k = j \land x = f\ i\ j)) \cup \{x. \exists k. (k \neq j \land x = 0)\}$
  by (simp only: Collect-disj-eq logic-aux2)
also have \dots = $\sum \{x. \exists k. (k = j \land x = f\ i\ j)\} + \sum \{x. \exists k. (k \neq j \land x = 0)\}$
  by (rule setsum-union, auto)
finally have $(f \otimes g) \ i\ j = f\ i\ j$
  by (auto simp add: setsum.neutral)

} 

thus ?thesis
by auto

qed

lemma mat-rearrange:
fixes $F :: 'a \Rightarrow 'k1 \Rightarrow 'k2 \Rightarrow 'b \Rightarrow 'c::doid-one-zero$
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 -
  {
    fix $z :: 'c$
    let $?lhs = \sum \{\{\sum \{ F\ i\ k1\ k2\ j | k2. k2 \in UNIV\} | k1. k1 \in UNIV\}$
    let $?rhs = \sum \{\{\sum \{ F\ i\ k1\ k2\ j | k1. k1 \in UNIV\} | k2. k2 \in UNIV\}$
    have $?lhs \leq z \iff (\forall k1 k2. F\ i\ k1\ k2\ j \leq z)$
      by (simp only: fset-to-im setsum-fun-image-sup fUNk1 fUNk2, auto)
    hence $?lhs \leq z \iff ?rhs \leq z$
      by (simp only: fset-to-im setsum-fun-image-sup fUNk1 fUNk2, auto)
  }

thus ?thesis
by (simp add: eq-iff)

qed

lemma mat-mult-assoc: $(f \otimes (g \otimes h)) = (f \otimes g) \otimes h$
proof -
  {
    fix $i\ j$
    have $(f \otimes (g \otimes h)) \ i\ j = \sum \{f\ i\ k1 \cdot \sum \{g\ k1\ k2 \cdot h\ k2\ j | k2. k2 \in UNIV\} | k1. k1 \in UNIV\}$
      by (simp only: mat-mult-def)
    also have \dots = $\sum \{\sum \{f\ i\ k1 \cdot g\ k1\ k2 \cdot h\ k2\ j | k2. k2 \in UNIV\} | k1. k1 \in UNIV\}$
      by (simp only: fset-to-im setsum-fun-distl finiteuniv mat.mult)
    also have \dots = $\sum \{\sum \{f\ i\ k1 \cdot g\ k1\ k2 \cdot h\ k2\ j | k1. k1 \in UNIV\} | k2. k2 \in UNIV\}$
      by (rule mat-rearrange, auto simp add: finiteuniv)
    also have \dots = $\sum \{\sum \{f\ i\ k1 \cdot g\ k1\ k2 | k1. k1 \in UNIV\} \cdot h\ k2\ j | k2. k2 \in UNIV\}$
  }

108
by (simp only: fset-to-im setsum-fun-distr finiteuniv)
finally have \((f \otimes (g \otimes h))\) \(i j = ((f \otimes g) \otimes h)\) \(i j\)
by (simp only: mat-mull-def)

}\nthus ?thesis
by auto
qed

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-less-eq \(f\) \(g\) ∧ \(f\) ≠ \(g\))

interpretation matrix-dioid: dioid-one-zero mat-add mat-mult mat-one mat-zero
mat-less-eq mat-less
by (unfold-locales) (metis mat-add-assoc mat-add-comm mat-mult-assoc [symmetric]
mat-distr mat-onel mat-oner mat zeror mat-annil mat-annir mat-less-eq-def mat-less-def
mat-add-idem mat-distl)+

As in the case of formal power series we currently do not implement the Kleene star of matrices, since this is complicated.

end

References


