

Partial Semigroups and Convolution Algebras

Brijesh Dongol, Victor B F Gomes, Ian J Hayes and Georg Struth

December 14, 2021

Abstract

Partial Semigroups are relevant to the foundations of quantum mechanics and combinatorics as well as to interval and separation logics. Convolution algebras can be understood either as algebras of generalised binary modalities over ternary Kripke frames, in particular over partial semigroups, or as algebras of quantale-valued functions which are equipped with a convolution-style operation of multiplication that is parametrised by a ternary relation. Convolution algebras provide algebraic semantics for various substructural logics, including categorial, relevance and linear logics, for separation logic and for interval logics; they cover quantitative and qualitative applications. These mathematical components for partial semigroups and convolution algebras provide uniform foundations from which models of computation based on relations, program traces or pomsets, and verification components for separation or interval temporal logics can be built with little effort.

Contents

1	Introductory Remarks	2
2	Partial Semigroups	3
2.1	Partial Semigroups	3
2.2	Green's Preorders and Green's Relations	3
2.3	Morphisms	5
2.4	Locally Finite Partial Semigroups	5
2.5	Cancellative Partial Semigroups	5
2.6	Partial Monoids	6
2.7	Cancellative Partial Monoids	7
2.8	Positive Partial Monoids	8
2.9	Positive Cancellative Partial Monoids	8
2.10	From Partial Abelian Semigroups to Partial Abelian Monoids	8
2.11	Alternative Definitions	9
2.12	Product Constructions	10
2.13	Partial Semigroup Actions and Semidirect Products	12
3	Models of Partial Semigroups	13
3.1	Partial Monoids of Segments and Intervals	13
3.2	Cancellative PAM's of Partial Functions	15
3.3	PAM's of Disjoint Unions of Sets	15

4	Quantales	16
4.1	Properties of Complete Lattices	16
4.2	Families of Proto-Quantales	17
4.3	Families of Quantales	19
4.4	Quantales of Booleans and Complete Boolean Algebras	20
4.5	Products of Quantales	21
4.6	Quantale Modules and Semidirect Products	22
5	Binary Modalities and Relational Convolution	26
5.1	Auxiliary Properties	26
5.2	Binary Modalities	27
5.3	Relational Convolution and Correspondence Theory	29
5.4	Lifting to Function Spaces	33
6	Unary Modalities	34
6.1	Forward and Backward Diamonds	34
6.2	Forward and Backward Boxes	35
6.3	Symmetries and Dualities	35
7	Liftings of Partial Semigroups	37
7.1	Relational Semigroups and Partial Semigroups	37
7.2	Liftings of Partial Abelian Semigroups	38

1 Introductory Remarks

These mathematical components supply formal proofs for two articles on *Convolution Algebras* [3] and *Convolution as a Unifying Concept* [2]. They are sparsely documented and referenced; additional information can be found in these articles, and in particular the first one.

The approach generalises previous Isabelle components for convolution algebras that were intended for separation logic and used partial abelian semigroups and monoids for modelling store-heap pairs [1]. Due to the applications in separation logic, a detailed account of cancellative and positive partial abelian monoids has been included, as these structures characterise the heap succinctly. Isabelle verification components based on this approach will be submitted as a separate AFP entry.

Our article on convolution algebras [3] provides a detailed account of convolution-based semantics for Halpern-Shoham-style interval logics [4, 7], interval temporal logics [6] and duration calculi [8] based on partial monoids. While general approaches, including modal algebras over semi-infinite intervals, are supported by the mathematical components provided, additional work on store models and assignments of variables to values is needed in order to build verification components for such interval logics.

Convolution-based liftings of partial semigroups of graphs and partial orders allow formalisations of models of true concurrency such as pomset languages and concurrent Kleene algebras [5] in Isabelle, too. An AFP entry for these is in preparation.

In all these approaches, the main task is to construct suitable partial semigroups or monoids of the computational models intended, for instance, closed intervals over the reals under fusion product, unions of heaplets (i.e. partial functions) provided their domains are disjoint, disjoint unions of graphs as parallel products. Our approach then allows a generic lifting to convolution algebras on suitable function spaces with algebraic properties, for instance of heaplets to the assertion algebra of separation logic with separating conjunction as convolution [1, 2], or

of intervals to algebraic counterparts of interval temporal logics or duration calculi with the chop operation as convolution [3]. We believe that this general construction supports other applications as well—qualitative and quantitative ones.

We would like to thank Alasdair Armstrong for his help with some Isabelle proofs and Tony Hoare for many discussions that helped us shaping the general approach.

2 Partial Semigroups

```
theory Partial-Semigroups
  imports Main
```

```
begin
```

```
notation times (infixl  $\cdot$  70)
and times (infixl  $\oplus$  70)
```

2.1 Partial Semigroups

In this context, partiality is modelled by a definedness constraint D instead of a bottom element, which would make the algebra total. This is common practice in mathematics.

```
class partial-times = times +
  fixes  $D :: 'a \Rightarrow 'a \Rightarrow bool$ 
```

The definedness constraints for associativity state that the right-hand side of the associativity law is defined if and only if the left-hand side is and that, in this case, both sides are equal. This and slightly different constraints can be found in the literature.

```
class partial-semigroup = partial-times +
  assumes add-assocD:  $D\ y\ z \wedge D\ x\ (y \cdot z) \longleftrightarrow D\ x\ y \wedge D\ (x \cdot y)\ z$ 
  and add-assoc:  $D\ x\ y \wedge D\ (x \cdot y)\ z \Longrightarrow (x \cdot y) \cdot z = x \cdot (y \cdot z)$ 
```

Every semigroup is a partial semigroup.

```
sublocale semigroup-mult  $\subseteq$  sg: partial-semigroup -  $\lambda x\ y.$  True
  by standard (simp-all add: mult-assoc)
```

```
context partial-semigroup
begin
```

The following abbreviation is useful for sublocale statements.

```
abbreviation (input)  $R\ x\ y\ z \equiv D\ y\ z \wedge x = y \cdot z$ 
```

```
lemma add-assocD-var1:  $D\ y\ z \wedge D\ x\ (y \cdot z) \Longrightarrow D\ x\ y \wedge D\ (x \cdot y)\ z$ 
  by (simp add: add-assocD)
```

```
lemma add-assocD-var2:  $D\ x\ y \wedge D\ (x \cdot y)\ z \Longrightarrow D\ y\ z \wedge D\ x\ (y \cdot z)$ 
  by (simp add: add-assocD)
```

```
lemma add-assoc-var:  $D\ y\ z \wedge D\ x\ (y \cdot z) \Longrightarrow (x \cdot y) \cdot z = x \cdot (y \cdot z)$ 
  by (simp add: add-assoc add-assocD)
```

2.2 Green's Preorders and Green's Relations

We define the standard Green's preorders and Green's relations. They are usually defined on monoids. On (partial) semigroups, we only obtain transitive relations.

definition $gR\text{-rel} :: 'a \Rightarrow 'a \Rightarrow \text{bool}$ (**infix** \preceq_R 50) **where**

$$x \preceq_R y = (\exists z. D\ x\ z \wedge x \cdot z = y)$$

definition $strict\text{-}gR\text{-rel} :: 'a \Rightarrow 'a \Rightarrow \text{bool}$ (**infix** \prec_R 50) **where**

$$x \prec_R y = (x \preceq_R y \wedge \neg y \preceq_R x)$$

definition $gL\text{-rel} :: 'a \Rightarrow 'a \Rightarrow \text{bool}$ (**infix** \preceq_L 50) **where**

$$x \preceq_L y = (\exists z. D\ z\ x \wedge z \cdot x = y)$$

definition $strict\text{-}gL\text{-rel} :: 'a \Rightarrow 'a \Rightarrow \text{bool}$ (**infix** \prec_L 50) **where**

$$x \prec_L y = (x \preceq_L y \wedge \neg y \preceq_L x)$$

definition $gH\text{-rel} :: 'a \Rightarrow 'a \Rightarrow \text{bool}$ (**infix** \preceq_H 50) **where**

$$x \preceq_H y = (x \preceq_L y \wedge x \preceq_R y)$$

definition $gJ\text{-rel} :: 'a \Rightarrow 'a \Rightarrow \text{bool}$ (**infix** \preceq_J 50) **where**

$$x \preceq_J y = (\exists v\ w. D\ v\ x \wedge D\ (v \cdot x)\ w \wedge (v \cdot x) \cdot w = y)$$

definition $gR\ x\ y = (x \preceq_R y \wedge y \preceq_R x)$

definition $gL\ x\ y = (x \preceq_L y \wedge y \preceq_L x)$

definition $gH\ x\ y = (x \preceq_H y \wedge y \preceq_H x)$

definition $gJ\ x\ y = (x \preceq_J y \wedge y \preceq_J x)$

definition $gR\text{-downset} :: 'a \Rightarrow 'a\ \text{set}$ (\downarrow [100]100) **where**

$$x \downarrow \equiv \{y. y \preceq_R x\}$$

The following counterexample rules out reflexivity.

lemma $x \preceq_R x$

oops

lemma $gR\text{-rel-trans}: x \preceq_R y \Longrightarrow y \preceq_R z \Longrightarrow x \preceq_R z$

by (*metis* $gR\text{-rel-def}$ $add\text{-assoc}$ $add\text{-assocD-var2}$)

lemma $gL\text{-rel-trans}: x \preceq_L y \Longrightarrow y \preceq_L z \Longrightarrow x \preceq_L z$

by (*metis* $gL\text{-rel-def}$ $add\text{-assocD-var1}$ $add\text{-assoc-var}$)

lemma $gR\text{-add-isol}: D\ z\ y \Longrightarrow x \preceq_R y \Longrightarrow z \cdot x \preceq_R z \cdot y$

apply (*simp* $add: gR\text{-rel-def}$)

using $add\text{-assocD-var1}$ $add\text{-assoc-var}$ **by** *blast*

lemma $gL\text{-add-isol}: D\ y\ z \Longrightarrow x \preceq_L y \Longrightarrow x \cdot z \preceq_L y \cdot z$

apply (*simp* $add: gL\text{-rel-def}$)

by (*metis* $add\text{-assoc}$ $add\text{-assocD-var2}$)

definition $annil :: 'a \Rightarrow \text{bool}$ **where**

$$annil\ x = (\forall y. D\ x\ y \wedge x \cdot y = x)$$

definition $annir :: 'a \Rightarrow \text{bool}$ **where**

$$annir\ x = (\forall y. D\ y\ x \wedge y \cdot x = x)$$

end

2.3 Morphisms

definition *ps-morphism* :: ('a::partial-semigroup ⇒ 'b::partial-semigroup) ⇒ bool **where**
ps-morphism f = (∀ x y. D x y ⇒ D (f x) (f y) ∧ f (x · y) = (f x) · (f y))

definition *strong-ps-morphism* :: ('a::partial-semigroup ⇒ 'b::partial-semigroup) ⇒ bool **where**
strong-ps-morphism f = (ps-morphism f ∧ (∀ x y. D (f x) (f y) ⇒ D x y))

2.4 Locally Finite Partial Semigroups

In locally finite partial semigroups, elements can only be split in finitely many ways.

class *locally-finite-partial-semigroup* = *partial-semigroup* +
assumes *loc-fin*: *finite* (x↓)

2.5 Cancellative Partial Semigroups

class *cancellative-partial-semigroup* = *partial-semigroup* +
assumes *add-cancel*: D z x ⇒ D z y ⇒ z · x = z · y ⇒ x = y
and *add-cancr*: D x z ⇒ D y z ⇒ x · z = y · z ⇒ x = y

begin

lemma *unique-resl*: D x z ⇒ D x z' ⇒ x · z = y ⇒ x · z' = y ⇒ z = z'
by (*simp add: add-cancel*)

lemma *unique-resr*: D z x ⇒ D z' x ⇒ z · x = y ⇒ z' · x = y ⇒ z = z'
by (*simp add: add-cancr*)

lemma *gR-rel-mult*: D x y ⇒ x ≤_R x · y
using *gR-rel-def* **by** *force*

lemma *gL-rel-mult*: D x y ⇒ y ≤_L x · y
using *gL-rel-def* **by** *force*

By cancellation, the element z is uniquely defined for each pair x y, provided it exists. In both cases, z is therefore a function of x and y; it is a quotient or residual of x y.

lemma *quotr-unique*: x ≤_R y ⇒ (∃!z. D x z ∧ y = x · z)
using *gR-rel-def add-cancel* **by** *force*

lemma *quotl-unique*: x ≤_L y ⇒ (∃!z. D z x ∧ y = z · x)
using *gL-rel-def unique-resr* **by** *force*

definition *rquot* y x = (*THE* z. D x z ∧ x · z = y)

definition *lquot* y x = (*THE* z. D z x ∧ z · x = y)

lemma *rquot-prop*: D x z ∧ y = x · z ⇒ z = *rquot* y x
by (*metis (mono-tags, lifting) rquot-def the-equality unique-resl*)

lemma *rquot-mult*: x ≤_R y ⇒ z = *rquot* y x ⇒ x · z = y
using *gR-rel-def rquot-prop* **by** *force*

lemma *rquot-D*: x ≤_R y ⇒ z = *rquot* y x ⇒ D x z
using *gR-rel-def rquot-prop* **by** *force*

lemma *add-rquot*: $x \preceq_R y \implies (D x z \wedge x \oplus z = y \longleftrightarrow z = \text{rquot } y x)$
using *gR-rel-def rquot-prop* **by** *fastforce*

lemma *add-canc1*: $D x y \implies \text{rquot } (x \cdot y) x = y$
using *rquot-prop* **by** *simp*

lemma *add-canc2*: $x \preceq_R y \implies x \cdot (\text{rquot } y x) = y$
using *gR-rel-def add-canc1* **by** *force*

lemma *add-canc2-prop*: $x \preceq_R y \implies \text{rquot } y x \preceq_L y$
using *gL-rel-mult rquot-D rquot-mult* **by** *fastforce*

The next set of lemmas establishes standard Galois connections for cancellative partial semi-groups.

lemma *gR-galois-imp1*: $D x z \implies x \cdot z \preceq_R y \implies z \preceq_R \text{rquot } y x$
by (*metis gR-rel-def add-assoc add-assocD-var2 rquot-prop*)

lemma *gR-galois-imp21*: $x \preceq_R y \implies z \preceq_R \text{rquot } y x \implies x \cdot z \preceq_R y$
using *gR-add-isol rquot-D rquot-mult* **by** *fastforce*

lemma *gR-galois-imp22*: $x \preceq_R y \implies z \preceq_R \text{rquot } y x \implies D x z$
using *gR-rel-def add-assocD add-canc1* **by** *fastforce*

lemma *gR-galois*: $x \preceq_R y \implies (D x z \wedge x \cdot z \preceq_R y \longleftrightarrow z \preceq_R \text{rquot } y x)$
using *gR-galois-imp1 gR-galois-imp21 gR-galois-imp22* **by** *blast*

lemma *gR-rel-defined*: $x \preceq_R y \implies D x (\text{rquot } y x)$
by (*simp add: rquot-D*)

lemma *ex-add-galois*: $D x z \implies (\exists y. x \cdot z = y \longleftrightarrow \text{rquot } y x = z)$
using *add-canc1* **by** *force*

end

2.6 Partial Monoids

We allow partial monoids with multiple units. This is similar to and inspired by small categories.

```
class partial-monoid = partial-semigroup +
  fixes E :: 'a set
  assumes unitl-ex:  $\exists e \in E. D e x \wedge e \cdot x = x$ 
  and unitr-ex:  $\exists e \in E. D x e \wedge x \cdot e = x$ 
  and units-eq:  $e1 \in E \implies e2 \in E \implies D e1 e2 \implies e1 = e2$ 
```

Every monoid is a partial monoid.

```
sublocale monoid-mult  $\subseteq$  mon: partial-monoid -  $\lambda x y. \text{True } \{1\}$ 
by (standard; simp-all)
```

```
context partial-monoid
begin
```

```
lemma units-eq-var:  $e1 \in E \implies e2 \in E \implies e1 \neq e2 \implies \neg D e1 e2$ 
using units-eq by force
```

In partial monoids, Green's relations become preorders, but need not be partial orders.

```

sublocale gR: preorder gR-rel strict-gR-rel
  apply standard
  apply (simp add: strict-gR-rel-def)
  using gR-rel-def unitr-ex apply force
  using gR-rel-trans by blast

```

```

sublocale gL: preorder gL-rel strict-gL-rel
  apply standard
  apply (simp add: strict-gL-rel-def)
  using gL-rel-def unitl-ex apply force
  using gL-rel-trans by blast

```

```

lemma  $x \preceq_R y \implies y \preceq_R x \implies x = y$ 
oops

```

```

lemma  $\text{annil } x \implies \text{annil } y \implies x = y$ 
oops

```

```

lemma  $\text{annir } x \implies \text{annir } y \implies x = y$ 
oops

```

end

Next we define partial monoid morphisms.

```

definition pm-morphism :: ('a::partial-monoid  $\Rightarrow$  'b::partial-monoid)  $\Rightarrow$  bool where
  pm-morphism f = (ps-morphism f  $\wedge$  ( $\forall e. e \in E \longrightarrow (f e) \in E$ ))

```

```

definition strong-pm-morphism :: ('a::partial-monoid  $\Rightarrow$  'b::partial-monoid)  $\Rightarrow$  bool where
  strong-pm-morphism f = (pm-morphism f  $\wedge$  ( $\forall e. (f e) \in E \longrightarrow e \in E$ ))

```

Partial Monoids with a single unit form a special case.

```

class partial-monoid-one = partial-semigroup + one +
  assumes oneDl:  $D x 1$ 
  and oneDr:  $D 1 x$ 
  and oner:  $x \cdot 1 = x$ 
  and onel:  $1 \cdot x = x$ 

```

begin

```

sublocale pmo: partial-monoid - - {1}
  by standard (simp-all add: oneDr onel oneDl oner)

```

end

2.7 Cancellative Partial Monoids

```

class cancellative-partial-monoid = cancellative-partial-semigroup + partial-monoid

```

begin

```

lemma canc-unitr:  $D x e \implies x \cdot e = x \implies e \in E$ 
  by (metis add-cancl unitr-ex)

```

```

lemma canc-unitl:  $D e x \implies e \cdot x = x \implies e \in E$ 

```

by (*metis add-cancr unitl-ex*)

end

2.8 Positive Partial Monoids

class *positive-partial-monoid* = *partial-monoid* +
 assumes *posl*: $D x y \implies x \cdot y \in E \implies x \in E$
 and *posr*: $D x y \implies x \cdot y \in E \implies y \in E$

begin

lemma *pos-unitl*: $D x y \implies e \in E \implies x \cdot y = e \implies x = e$
 by (*metis posl posr unitr-ex units-eq-var*)

lemma *pos-unitr*: $D x y \implies e \in E \implies x \cdot y = e \implies y = e$
 by (*metis posl posr unitr-ex units-eq-var*)

end

2.9 Positive Cancellative Partial Monoids

class *positive-cancellative-partial-monoid* = *positive-partial-monoid* + *cancellative-partial-monoid*

begin

In positive cancellative monoids, the Green's relations are partial orders.

sublocale *pcpmR*: *order gR-rel strict-gR-rel*
 apply *standard*
 apply (*clarsimp simp: gR-rel-def*)
 by (*metis canc-unitr add-assoc add-assocD-var2 pos-unitl*)

sublocale *pcpmL*: *order gL-rel strict-gL-rel*
 apply *standard*
 apply (*clarsimp simp: gL-rel-def*)
 by (*metis canc-unitl add-assoc add-assocD-var1 pos-unitr*)

end

2.10 From Partial Abelian Semigroups to Partial Abelian Monoids

Next we define partial abelian semigroups. These are interesting, e.g., for the foundations of quantum mechanics and as resource monoids in separation logic.

class *pas* = *partial-semigroup* +
 assumes *add-comm*: $D x y \implies D y x \wedge x \oplus y = y \oplus x$

begin

lemma *D-comm*: $D x y \iff D y x$
 by (*auto simp add: add-comm*)

lemma *add-comm'*: $D x y \implies x \oplus y = y \oplus x$
 by (*auto simp add: add-comm*)

lemma *gL-gH-rel*: $(x \preceq_L y) = (x \preceq_H y)$

apply (*simp add: gH-rel-def gL-rel-def gR-rel-def*)
using *add-comm* **by** *force*

lemma *gR-gH-rel*: $(x \preceq_R y) = (x \preceq_H y)$
apply (*simp add: gH-rel-def gL-rel-def gR-rel-def*)
using *add-comm* **by** *blast*

lemma *annilr*: $\text{annil } x = \text{annir } x$
by (*metis annil-def annir-def add-comm*)

lemma *anni-unique*: $\text{annil } x \implies \text{annil } y \implies x = y$
by (*metis annilr annil-def annir-def*)

end

The following classes collect families of partially ordered abelian semigroups and monoids.

class *locally-finite-pas* = *pas* + *locally-finite-partial-semigroup*

class *pam* = *pas* + *partial-monoid*

class *cancellative-pam* = *pam* + *cancellative-partial-semigroup*

class *positive-pam* = *pam* + *positive-partial-monoid*

class *positive-cancellative-pam* = *positive-pam* + *cancellative-pam*

class *generalised-effect-algebra* = *pas* + *partial-monoid-one*

class *cancellative-pam-one* = *cancellative-pam* + *partial-monoid-one*

class *positive-cancellative-pam-one* = *positive-cancellative-pam* + *cancellative-pam-one*

context *cancellative-pam-one*

begin

lemma *E-eq-one*: $E = \{1\}$
by (*metis oneDr oner unitl-ex units-eq singleton-iff subsetI subset-antisym*)

lemma *one-in-E*: $1 \in E$
by (*simp add: E-eq-one*)

end

2.11 Alternative Definitions

PAS's can be axiomatised more compactly as follows.

class *pas-alt* = *partial-times* +
assumes *pas-alt-assoc*: $D x y \wedge D (x \oplus y) z \implies D y z \wedge D x (y \oplus z) \wedge (x \oplus y) \oplus z = x \oplus (y \oplus z)$
and *pas-alt-comm*: $D x y \implies D y x \wedge x \oplus y = y \oplus x$

sublocale *pas-alt* \subseteq *palt*: *pas*
apply *standard*
using *pas-alt-assoc pas-alt-comm* **by** *blast+*

Positive abelian PAM's can be axiomatised more compactly as well.

class *pam-pos-alt* = *pam* +
assumes *pos-alt*: $D\ x\ y \implies e \in E \implies x \oplus y = e \implies x = e$

sublocale *pam-pos-alt* \subseteq *ppalt*: *positive-pam*
apply *standard*
using *pos-alt* **apply** *force*
using *add-comm pos-alt* **by** *fastforce*

2.12 Product Constructions

We consider two kinds of product construction. The first one combines partial semigroups with sets, the second one partial semigroups with partial semigroups. The first one is interesting for Separation Logic. Semidirect product constructions are considered later.

instantiation *prod* :: (*type*, *partial-semigroup*) *partial-semigroup*
begin

definition *D-prod* $x\ y = (fst\ x = fst\ y \wedge D\ (snd\ x)\ (snd\ y))$
for $x\ y :: 'a \times 'b$

definition *times-prod* :: $'a \times 'b \Rightarrow 'a \times 'b \Rightarrow 'a \times 'b$ **where**
times-prod $x\ y = (fst\ x, snd\ x \cdot snd\ y)$

instance
apply (*standard*, *simp-all add: D-prod-def times-prod-def*)
using *partial-semigroup-class.add-assocD* **apply** *force*
by (*simp add: partial-semigroup-class.add-assoc*)

end

instantiation *prod* :: (*type*, *partial-monoid*) *partial-monoid*
begin

definition *E-prod* :: $('a \times 'b)$ *set* **where**
E-prod = $\{x. snd\ x \in E\}$

instance
apply (*standard*, *simp-all add: D-prod-def times-prod-def E-prod-def*)
using *partial-monoid-class.unitl-ex* **apply** *fastforce*
using *partial-monoid-class.unitr-ex* **apply** *fastforce*
by (*simp add: partial-monoid-class.units-eq prod-eq-iff*)

end

instance *prod* :: (*type*, *pas*) *pas*
apply (*standard*, *simp add: D-prod-def times-prod-def*)
using *pas-class.add-comm* **by** *force*

lemma *prod-div1*: $(x1 :: 'a, y1 :: 'b::pas) \preceq_R (x2 :: 'a, y2 :: 'b::pas) \implies x1 = x2$
by (*force simp: partial-semigroup-class.gR-rel-def times-prod-def*)

lemma *prod-div2*: $(x1, y1) \preceq_R (x2, y2) \implies y1 \preceq_R y2$
by (*force simp: partial-semigroup-class.gR-rel-def D-prod-def times-prod-def*)

lemma *prod-div-eq*: $(x1, y1) \preceq_R (x2, y2) \longleftrightarrow x1 = x2 \wedge y1 \preceq_R y2$
by (*force simp: partial-semigroup-class.gR-rel-def D-prod-def times-prod-def*)

```

instance prod :: (type, pam) pam
  by standard

instance prod :: (type, cancellative-pam) cancellative-pam
  by (standard, auto simp: D-prod-def times-prod-def add-cancr add-cancl)

lemma prod-res-eq: (x1, y1)  $\preceq_R$  (x2::'a,y2::'b::cancellative-pam)
   $\implies$  rquot (x2, y2) (x1, y1) = (x1, rquot y2 y1)
apply (clarsimp simp: partial-semigroup-class.gR-rel-def D-prod-def times-prod-def rquot-def)
apply (rule theI2 conjI)
  apply force
using add-cancl apply force
by (rule the-equality, auto simp: add-cancl)

instance prod :: (type, positive-pam) positive-pam
  apply (standard, simp-all add: E-prod-def D-prod-def times-prod-def)
using positive-partial-monoid-class.posl apply blast
using positive-partial-monoid-class.posr by blast

instance prod :: (type, positive-cancellative-pam) positive-cancellative-pam ..

instance prod :: (type, locally-finite-pas) locally-finite-pas
proof (standard, case-tac x, clarsimp)
  fix s :: 'a and x :: 'b
  have finite (x $\downarrow$ )
    by (simp add: loc-fin)
  hence finite {y.  $\exists z. D y z \wedge y \oplus z = x$ }
    by (simp add: partial-semigroup-class.gR-downset-def partial-semigroup-class.gR-rel-def)
  hence finite {(s, y) | y.  $\exists z. D y z \wedge y \oplus z = x$ }
    by (drule-tac f= $\lambda y. (s, y)$  in finite-image-set)
  moreover have {y.  $\exists z1 z2. D y (z1, z2) \wedge y \oplus (z1, z2) = (s, x)$ }
     $\subseteq$  {(s, y) | y.  $\exists z. D y z \wedge y \oplus z = x$ }
    by (auto simp: D-prod-def times-prod-def)
  ultimately have finite {y.  $\exists z1 z2. D y (z1, z2) \wedge y \oplus (z1, z2) = (s, x)$ }
    by (auto intro: finite-subset)
  thus finite ((s, x) $\downarrow$ )
    by (simp add: partial-semigroup-class.gR-downset-def partial-semigroup-class.gR-rel-def)
qed

Next we consider products of two partial semigroups.

definition ps-prod-D :: 'a :: partial-semigroup  $\times$  'b :: partial-semigroup  $\Rightarrow$  'a  $\times$  'b  $\Rightarrow$  bool
  where ps-prod-D x y  $\equiv$  D (fst x) (fst y)  $\wedge$  D (snd x) (snd y)

definition ps-prod-times :: 'a :: partial-semigroup  $\times$  'b :: partial-semigroup  $\Rightarrow$  'a  $\times$  'b  $\Rightarrow$  'a  $\times$  'b
  where ps-prod-times x y = (fst x  $\cdot$  fst y, snd x  $\cdot$  snd y)

interpretation ps-prod: partial-semigroup ps-prod-times ps-prod-D
  apply (standard, simp-all add: ps-prod-D-def ps-prod-times-def)
  apply (meson partial-semigroup-class.add-assocD)
  by (simp add: partial-semigroup-class.add-assoc)

interpretation pas-prod: pas ps-prod-times ps-prod-D :: 'a :: pas  $\times$  'b :: pas  $\Rightarrow$  'a  $\times$  'b  $\Rightarrow$  bool
  by (standard, clarsimp simp: ps-prod-D-def ps-prod-times-def pas-class.add-comm)

```

definition *pm-prod-E* :: ('a :: partial-monoid × 'b :: partial-monoid) set **where**
pm-prod-E = {x. fst x ∈ E ∧ snd x ∈ E}

interpretation *pm-prod*: partial-monoid ps-prod-times ps-prod-D pm-prod-E
apply standard
apply (simp-all add: ps-prod-times-def ps-prod-D-def pm-prod-E-def)
apply (metis partial-monoid-class.unitl-ex prod.collapse)
apply (metis partial-monoid-class.unitr-ex prod.collapse)
by (simp add: partial-monoid-class.units-eq prod.expand)

interpretation *pam-prod*: pam ps-prod-times ps-prod-D pm-prod-E :: ('a :: pam × 'a :: pam) set ..

2.13 Partial Semigroup Actions and Semidirect Products

(Semi)group actions are a standard mathematical construction. We generalise this to partial semigroups and monoids. We use it to define semidirect products of partial semigroups. A generalisation to wreath products might be added in the future.

First we define the (left) action of a partial semigroup on a set. A right action could be defined in a similar way, but we do not pursue this at the moment.

locale *partial-sg-laction* =
fixes *Dla* :: 'a::partial-semigroup ⇒ 'b ⇒ bool
and *act* :: 'a::partial-semigroup ⇒ 'b ⇒ 'b (α)
assumes *act-assocD*: D x y ∧ *Dla* (x · y) p ⟷ *Dla* y p ∧ *Dla* x (α y p)
and *act-assoc*: D x y ∧ *Dla* (x · y) p ⟹ α (x · y) p = α x (α y p)

Next we define the action of a partial semigroup on another partial semigroup. In the tradition of semigroup theory we use addition as a non-commutative operation for the second semigroup.

locale *partial-sg-sg-laction* = *partial-sg-laction* +
assumes *act-distribD*: D (p::'b::partial-semigroup) q ∧ *Dla* (x::'a::partial-semigroup) (p ⊕ q) ⟷ *Dla* x p ∧ *Dla* x q ∧ D (α x p) (α x q)
and *act-distrib*: D p q ∧ *Dla* x (p ⊕ q) ⟹ α x (p ⊕ q) = (α x p) ⊕ (α x q)

begin

Next we define the semidirect product as a partial operation and show that the semidirect product of two partial semigroups forms a partial semigroup.

definition *sd-D* :: ('a × 'b) ⇒ ('a × 'b) ⇒ bool **where**
sd-D x y ≡ D (fst x) (fst y) ∧ *Dla* (fst x) (snd y) ∧ D (snd x) (α (fst x) (snd y))

definition *sd-prod* :: ('a × 'b) ⇒ ('a × 'b) ⇒ ('a × 'b) **where**
sd-prod x y = ((fst x) · (fst y), (snd x) ⊕ (α (fst x) (snd y)))

sublocale *dp-semigroup*: partial-semigroup *sd-prod* *sd-D*
apply *unfold-locales*
apply (simp-all add: *sd-prod-def* *sd-D-def*)
apply (clarsimp, metis *act-assoc* *act-assocD* *act-distrib* *act-distribD* *add-assocD*)
by (clarsimp, metis *act-assoc* *act-assocD* *act-distrib* *act-distribD* *add-assoc* *add-assocD*)

end

Finally we define the semigroup action for two partial monoids and show that the semidirect product of two partial monoids is a partial monoid.

locale *partial-mon-sg-laction* = *partial-sg-sg-laction* *Dla*

```

for Dla :: 'a::partial-monoid  $\Rightarrow$  'b::partial-semigroup  $\Rightarrow$  bool +
assumes act-unitl:  $e \in E \Longrightarrow Dla\ e\ p \wedge \alpha\ e\ p = p$ 

locale partial-mon-mon-laction = partial-mon-sg-laction - Dla
for Dla :: 'a::partial-monoid  $\Rightarrow$  'b::partial-monoid  $\Rightarrow$  bool +
assumes act-annir:  $e \in Ea \Longrightarrow Dla\ x\ e \wedge \alpha\ x\ e = e$ 

begin

definition sd-E :: ('a  $\times$  'b) set where
  sd-E = {x. fst x  $\in E \wedge$  snd x  $\in E$ }

sublocale dp-semigroup : partial-monoid sd-prod sd-D sd-E
apply unfold-locales
  apply (simp-all add: sd-prod-def sd-D-def sd-E-def)
  apply (metis act-annir eq-fst-iff eq-snd-iff mem-Collect-eq partial-monoid-class.unitl-ex)
  apply (metis act-annir eq-fst-iff eq-snd-iff partial-monoid-class.unitr-ex)
by (metis act-annir partial-monoid-class.units-eq prod-eqI)

end

end

```

3 Models of Partial Semigroups

```

theory Partial-Semigroup-Models
imports Partial-Semigroups

```

```

begin

```

So far this section collects three models that we need for applications. Other interesting models might be added in the future. These might include binary relations, formal power series and matrices, paths in graphs under fusion, program traces with alternating state and action symbols under fusion, partial orders under series and parallel products.

3.1 Partial Monoids of Segments and Intervals

Segments of a partial order are sub partial orders between two points. Segments generalise intervals in that intervals are segments in linear orders. We formalise segments and intervals as pairs, where the first coordinate is smaller than the second one. Algebras of segments and intervals are interesting in Rota's work on the foundations of combinatorics as well as for interval logics and duration calculi.

First we define the subtype of ordered pairs of one single type.

```

typedef 'a dprod = {(x::'a, y::'a). True}
by simp

```

```

setup-lifting type-definition-dprod

```

Such pairs form partial semigroups and partial monoids with respect to fusion.

```

instantiation dprod :: (type) partial-semigroup
begin

```

lift-definition $D\text{-dprod} :: 'a \text{ dprod} \Rightarrow 'a \text{ dprod} \Rightarrow \text{bool}$ **is** $\lambda x y. (\text{snd } x = \text{fst } y)$.

lift-definition $\text{times-dprod} :: 'a \text{ dprod} \Rightarrow 'a \text{ dprod} \Rightarrow 'a \text{ dprod}$ **is** $\lambda x y. (\text{fst } x, \text{snd } y)$
by *simp*

instance

by *standard (transfer, force)+*

end

instantiation $\text{dprod} :: (\text{type}) \text{ partial-monoid}$

begin

lift-definition $E\text{-dprod} :: 'a \text{ dprod set}$ **is** $\{x. \text{fst } x = \text{snd } x\}$

by *simp*

instance

by *standard (transfer, force)+*

end

Next we define the type of segments.

typedef (**overloaded**) $'a \text{ segment} = \{x :: ('a :: \text{order} \times 'a :: \text{order}). \text{fst } x \leq \text{snd } x\}$

by *force*

setup-lifting *type-definition-segment*

Segments form partial monoids as well.

instantiation $\text{segment} :: (\text{order}) \text{ partial-monoid}$

begin

lift-definition $E\text{-segment} :: 'a \text{ segment set}$ **is** $\{x. \text{fst } x = \text{snd } x\}$

by *simp*

lift-definition $D\text{-segment} :: 'a :: \text{order} \text{ segment} \Rightarrow 'a \text{ segment} \Rightarrow \text{bool}$

is $\lambda x y. (\text{snd } x = \text{fst } y)$.

lift-definition $\text{times-segment} :: 'a :: \text{order} \text{ segment} \Rightarrow 'a \text{ segment} \Rightarrow 'a \text{ segment}$

is $\lambda x y. \text{if } \text{snd } x = \text{fst } y \text{ then } (\text{fst } x, \text{snd } y) \text{ else } x$

by *auto*

instance

by *standard (transfer, force)+*

end

Next we define the function *segm* that maps segments-as-pairs to segments-as-sets.

definition $\text{segm} :: 'a :: \text{order} \text{ segment} \Rightarrow 'a \text{ set}$ **where**

$\text{segm } x = \{y. \text{fst } (\text{Rep-segment } x) \leq y \wedge y \leq \text{snd } (\text{Rep-segment } x)\}$

thm *Rep-segment*

lemma *segm-sub-morph*: $\text{snd } (\text{Rep-segment } x) = \text{fst } (\text{Rep-segment } y) \implies \text{segm } x \cup \text{segm } y \leq \text{segm } (x \cdot y)$

apply (*simp add: segm-def times-segment.rep-eq, safe*)

using *Rep-segment dual-order.trans* **apply** *blast*
by (*metis (mono-tags, lifting) Rep-segment dual-order.trans mem-Collect-eq*)

The function `segm` is not generally a morphism.

lemma *snd (Rep-segment x) = fst (Rep-segment y) \implies segm x \cup segm y = segm (x \cdot y)*
oops

Intervals are segments over orders that satisfy Halpern and Shoham's linear order property. This is still more general than linearity of the poset.

class *lip-order = order +*
assumes *lip: x \leq y \implies ($\forall v w. (x \leq v \wedge v \leq y \wedge x \leq w \wedge w \leq y \longrightarrow v \leq w \vee w \leq v)$)*

The function `segm` is now a morphism.

lemma *segm-morph: snd (Rep-segment x::('a::lip-order \times 'a::lip-order)) = fst (Rep-segment y)*
 \implies *segm x \cup segm y = segm (x \cdot y)*
apply (*simp add: segm-def times-segment-def*)
apply (*transfer, clarsimp simp add: Abs-segment-inverse lip, safe*)
apply *force+*
by (*meson lip order-trans*)

3.2 Cancellative PAM's of Partial Functions

We show that partial functions under disjoint union form a positive cancellative PAM. This is interesting for modeling the heap in separation logic.

type-synonym *'a pfun = 'a \Rightarrow 'a option*

definition *ortho :: 'a pfun \Rightarrow 'a pfun \Rightarrow bool*
where *ortho f g \equiv dom f \cap dom g = {}*

lemma *pfun-comm: ortho x y \implies x ++ y = y ++ x*
by (*force simp: ortho-def intro!: map-add-comm*)

lemma *pfun-canc: ortho z x \implies ortho z y \implies z ++ x = z ++ y \implies x = y*
apply (*auto simp: ortho-def map-add-def option.case-eq-if fun-eq-iff*)
by (*metis domIff dom-restrict option.collapse restrict-map-def*)

interpretation *pfun: positive-cancellative-pam-one map-add ortho {Map.empty} Map.empty*
apply (*standard, auto simp: ortho-def pfun-canc*)
by (*simp-all add: inf-commute map-add-comm ortho-def pfun-canc*)

3.3 PAM's of Disjoint Unions of Sets

This simple disjoint union construction underlies important compositions of graphs or partial orders, in particular in the context of complete joins and disjoint unions of graphs and of series and parallel products of partial orders.

instantiation *set :: (type) pas*
begin

definition *D-set :: 'a set \Rightarrow 'a set \Rightarrow bool* **where**
D-set x y \equiv x \cap y = {}

definition *times-set :: 'a set \Rightarrow 'a set \Rightarrow 'a set* **where**
times-set x y = x \cup y

```

instance
  by standard (auto simp: D-set-def times-set-def)

end

instantiation set :: (type) pam
begin

definition E-set :: 'a set set where
  E-set = {{}}

instance
  by standard (auto simp: D-set-def times-set-def E-set-def)

end

end

```

4 Quantales

This entry will be merged eventually with other quantale entries and become a standalone one.

```

theory Quantales
  imports Main

```

```

begin

```

```

notation sup (infixl  $\sqcup$  60)
  and inf (infixl  $\sqcap$  55)
  and top ( $\top$ )
  and bot ( $\perp$ )
  and relcomp (infixl ; 70)
  and times (infixl  $\cdot$  70)
  and Sup ( $\bigsqcup$ - [900] 900)
  and Inf ( $\bigsqcap$ - [900] 900)

```

4.1 Properties of Complete Lattices

```

lemma (in complete-lattice) Sup-sup-pred:  $x \sqcup \bigsqcup \{y. P y\} = \bigsqcup \{y. y = x \vee P y\}$ 
  apply (rule order.antisym)
  apply (simp add: Collect-mono Sup-subset-mono Sup-upper)
  using Sup-least Sup-upper sup.coboundedI2 by force

```

```

lemma (in complete-lattice) sup-Sup:  $x \sqcup y = \bigsqcup \{x, y\}$ 
  by simp

```

```

lemma (in complete-lattice) sup-Sup-var:  $x \sqcup y = \bigsqcup \{z. z \in \{x, y\}\}$ 
  by (metis Collect-mem-eq sup-Sup)

```

```

lemma (in complete-boolean-algebra) shunt1:  $x \sqcap y \leq z \iff x \leq -y \sqcup z$ 
proof standard
  assume  $x \sqcap y \leq z$ 
  hence  $-y \sqcup (x \sqcap y) \leq -y \sqcup z$ 
  using sup.mono by blast

```



```

hence  $-y \sqcup x \leq -y \sqcup z$ 
  by (simp add: sup-inf-distrib1)
thus  $x \leq -y \sqcup z$ 
  by simp
next
assume  $x \leq -y \sqcup z$ 
hence  $x \sqcap y \leq (-y \sqcup z) \sqcap y$ 
  using inf-mono by auto
thus  $x \sqcap y \leq z$ 
  using inf.boundedE inf-sup-distrib2 by auto
qed

lemma (in complete-boolean-algebra) meet-shunt:  $x \sqcap y = \perp \iff x \leq -y$ 
  by (metis bot-least inf-absorb2 inf-compl-bot-left2 shunt1 sup-absorb1)

lemma (in complete-boolean-algebra) join-shunt:  $x \sqcup y = \top \iff -x \leq y$ 
  by (metis compl-sup compl-top-eq double-compl meet-shunt)

```

4.2 Families of Proto-Quantales

Proto-Quantales are complete lattices equipped with an operation of composition or multiplication that need not be associative.

```

class proto-near-quantale = complete-lattice + times +
  assumes Sup-distr:  $\bigsqcup X \cdot y = \bigsqcup \{x \cdot y \mid x. x \in X\}$ 

```

begin

```

lemma mult-botl [simp]:  $\perp \cdot x = \perp$ 
  using Sup-distr[where  $X=\{\}$ ] by auto

lemma sup-distr:  $(x \sqcup y) \cdot z = (x \cdot z) \sqcup (y \cdot z)$ 
  using Sup-distr[where  $X=\{x, y\}$ ] by (fastforce intro!: Sup-eqI)

lemma mult-isor:  $x \leq y \implies x \cdot z \leq y \cdot z$ 
  by (metis sup.absorb-iff1 sup-distr)

definition bres :: ' $a \Rightarrow 'a \Rightarrow 'a$  (infixr  $\rightarrow$  60) where
   $x \rightarrow z = \bigsqcup \{y. x \cdot y \leq z\}$ 

definition fres :: ' $a \Rightarrow 'a \Rightarrow 'a$  (infixl  $\leftarrow$  60) where
   $z \leftarrow y = \bigsqcup \{x. x \cdot y \leq z\}$ 

lemma bres-galois-imp:  $x \cdot y \leq z \longrightarrow y \leq x \rightarrow z$ 
  by (simp add: Sup-upper bres-def)

```

```

lemma fres-galois:  $x \cdot y \leq z \iff x \leq z \leftarrow y$ 

```

proof

```

  show  $x \cdot y \leq z \implies x \leq z \leftarrow y$ 
    by (simp add: Sup-upper fres-def)
next
assume  $x \leq z \leftarrow y$ 
hence  $x \cdot y \leq \bigsqcup \{x. x \cdot y \leq z\} \cdot y$ 
  by (simp add: fres-def mult-isor)
also have  $\dots = \bigsqcup \{x \cdot y \mid x. x \cdot y \leq z\}$ 
  by (simp add: Sup-distr)

```

```

    also have ... ≤ z
      by (rule Sup-least, auto)
    finally show x · y ≤ z .
qed

end

class proto-pre-quantale = proto-near-quantale +
  assumes Sup-subdistl:  $\bigsqcup \{x \cdot y \mid y \cdot y \in Y\} \leq x \cdot \bigsqcup Y$ 

begin

lemma sup-subdistl:  $(x \cdot y) \sqcup (x \cdot z) \leq x \cdot (y \sqcup z)$ 
  using Sup-subdistl[where Y={y, z}] Sup-le-iff by auto

lemma mult-isol:  $x \leq y \implies z \cdot x \leq z \cdot y$ 
  by (metis le-iff-sup le-sup-iff sup-subdistl)

end

class weak-proto-quantale = proto-near-quantale +
  assumes weak-Sup-distl:  $Y \neq \{\} \implies x \cdot \bigsqcup Y = \bigsqcup \{x \cdot y \mid y \cdot y \in Y\}$ 

begin

subclass proto-pre-quantale
proof standard
  have a:  $\bigwedge x Y. Y = \{\} \implies \bigsqcup \{x \cdot y \mid y \cdot y \in Y\} \leq x \cdot \bigsqcup Y$ 
    by simp
  have b:  $\bigwedge x Y. Y \neq \{\} \implies \bigsqcup \{x \cdot y \mid y \cdot y \in Y\} \leq x \cdot \bigsqcup Y$ 
    by (simp add: weak-Sup-distl)
  show  $\bigwedge x Y. \bigsqcup \{x \cdot y \mid y \cdot y \in Y\} \leq x \cdot \bigsqcup Y$ 
    using a b by blast
qed

lemma sup-distl:  $x \cdot (y \sqcup z) = (x \cdot y) \sqcup (x \cdot z)$ 
  using weak-Sup-distl[where Y={y, z}] by (fastforce intro!: Sup-eqI)

lemma y ≤ x → z → x · y ≤ z
oops

end

class proto-quantale = proto-near-quantale +
  assumes Sup-distl:  $x \cdot \bigsqcup Y = \bigsqcup \{x \cdot y \mid y \cdot y \in Y\}$ 

begin

subclass weak-proto-quantale
  by standard (simp add: Sup-distl)

lemma bres-galois:  $x \cdot y \leq z \iff y \leq x \rightarrow z$ 
proof
  show  $x \cdot y \leq z \implies y \leq x \rightarrow z$ 
    by (simp add: Sup-upper bres-def)

```

```

next
  assume  $y \leq x \rightarrow z$ 
  hence  $x \cdot y \leq x \cdot \bigsqcup \{y. x \cdot y \leq z\}$ 
    by (simp add: bres-def mult-isol)
  also have  $\dots = \bigsqcup \{x \cdot y \mid y. x \cdot y \leq z\}$ 
    by (simp add: Sup-distl)
  also have  $\dots \leq z$ 
    by (rule Sup-least, auto)
  finally show  $x \cdot y \leq z$  .
qed

end

```

4.3 Families of Quantales

```

class near-quantale = proto-near-quantale + semigroup-mult

```

```

class unital-near-quantale = near-quantale + monoid-mult

```

```

begin

```

```

definition iter :: 'a  $\Rightarrow$  'a where
  iter x  $\equiv$   $\prod \{y. \exists i. y = x \wedge i\}$ 

```

```

lemma iter-ref [simp]: iter x  $\leq 1$ 

```

```

  apply (simp add: iter-def)

```

```

  by (metis (mono-tags, lifting) Inf-lower local.power.power-0 mem-Collect-eq)

```

```

lemma le-top:  $x \leq \top \cdot x$ 

```

```

  by (metis mult-isor mult.monoid-axioms top-greatest monoid.left-neutral)

```

```

end

```

```

class pre-quantale = proto-pre-quantale + semigroup-mult

```

```

subclass (in pre-quantale) near-quantale ..

```

```

class unital-pre-quantale = pre-quantale + monoid-mult

```

```

subclass (in unital-pre-quantale) unital-near-quantale ..

```

```

class weak-quantale = weak-proto-quantale + semigroup-mult

```

```

subclass (in weak-quantale) pre-quantale ..

```

The following counterexample shows an important consequence of weakness: the absence of right annihilation.

```

lemma (in weak-quantale)  $x \cdot \perp = \perp$ 

```

```

  oops

```

```

class unital-weak-quantale = weak-quantale + monoid-mult

```

```

lemma (in unital-weak-quantale)  $x \cdot \perp = \perp$ 

```

```

  oops

```

```

subclass (in unital-weak-quantale) unital-pre-quantale ..

class quantale = proto-quantale + semigroup-mult

begin

subclass weak-quantale ..

lemma mult-botr [simp]:  $x \cdot \perp = \perp$ 
  using Sup-distl[where  $Y = \{\}$ ] by auto

end

class unital-quantale = quantale + monoid-mult

subclass (in unital-quantale) unital-weak-quantale ..

class ab-quantale = quantale + ab-semigroup-mult

begin

lemma bres-fres-eq:  $x \rightarrow y = y \leftarrow x$ 
  by (simp add: fres-def bres-def mult-commute)

end

class ab-unital-quantale = ab-quantale + unital-quantale

class distrib-quantale = quantale + complete-distrib-lattice

class bool-quantale = quantale + complete-boolean-algebra

class distrib-unital-quantale = unital-quantale + complete-distrib-lattice

class bool-unital-quantale = unital-quantale + complete-boolean-algebra

class distrib-ab-quantale = distrib-quantale + ab-quantale

class bool-ab-quantale = bool-quantale + ab-quantale

class distrib-ab-unital-quantale = distrib-quantale + unital-quantale

class bool-ab-unital-quantale = bool-ab-quantale + unital-quantale

```

4.4 Quantaes of Booleans and Complete Boolean Algebras

```

instantiation bool :: bool-ab-unital-quantale
begin

definition one-bool = True

definition times-bool = ( $\lambda x y. x \wedge y$ )

instance
  by standard (auto simp: times-bool-def one-bool-def)

```

end

context *complete-distrib-lattice*
begin

interpretation *cdl-quantale: distrib-quantale - - - - - inf*
by *standard (simp-all add: inf.assoc Setcompr-eq-image Sup-inf inf-Sup)*

end

context *complete-boolean-algebra*
begin

interpretation *cba-quantale: bool-ab-unital-quantale inf - - - - - top*
apply *standard*
apply (*simp add: inf.assoc*)
apply (*simp add: inf.commute*)
apply (*simp add: Setcompr-eq-image Sup-inf*)
apply (*simp add: Setcompr-eq-image inf-Sup*)
by *simp-all*

In this setting, residuation can be translated like classical implication.

lemma *cba-bres1: $x \sqcap y \leq z \iff x \leq \text{cba-quantale.bres } y \ z$*
using *cba-quantale.bres-galois inf.commute* **by** *fastforce*

lemma *cba-bres2: $x \leq -y \sqcup z \iff x \leq \text{cba-quantale.bres } y \ z$*
using *cba-bres1 shunt1* **by** *auto*

lemma *cba-bres-prop: $\text{cba-quantale.bres } x \ y = -x \sqcup y$*
using *cba-bres2 order.eq-iff* **by** *blast*

end

Other models will follow.

4.5 Products of Quantales

definition *Inf-prod* $X = (\sqcap \{fst \ x \mid x. \ x \in X\}, \sqcap \{snd \ x \mid x. \ x \in X\})$

definition *inf-prod* $x \ y = (fst \ x \sqcap fst \ y, snd \ x \sqcap snd \ y)$

definition *bot-prod* $= (bot, bot)$

definition *Sup-prod* $X = (\sqcup \{fst \ x \mid x. \ x \in X\}, \sqcup \{snd \ x \mid x. \ x \in X\})$

definition *sup-prod* $x \ y = (fst \ x \sqcup fst \ y, snd \ x \sqcup snd \ y)$

definition *top-prod* $= (top, top)$

definition *less-eq-prod* $x \ y \equiv less-eq \ (fst \ x) \ (fst \ y) \wedge less-eq \ (snd \ x) \ (snd \ y)$

definition *less-prod* $x \ y \equiv less-eq \ (fst \ x) \ (fst \ y) \wedge less-eq \ (snd \ x) \ (snd \ y) \wedge x \neq y$

definition *times-prod'* $x \ y = (fst \ x \cdot fst \ y, snd \ x \cdot snd \ y)$

definition *one-prod* = (1,1)

interpretation *prod*: complete-lattice Inf-prod Sup-prod inf-prod less-eq-prod less-prod sup-prod bot-prod
top-prod :: ('a::complete-lattice × 'b::complete-lattice)

apply *standard*

apply (*simp-all add*: Inf-prod-def Sup-prod-def inf-prod-def sup-prod-def bot-prod-def top-prod-def
less-eq-prod-def less-prod-def Sup-distl Sup-distr)

apply *force+*

apply (*rule conjI*, (*rule Inf-lower*, *force*)+)

apply (*rule conjI*, (*rule Inf-greatest*, *force*)+)

apply (*rule conjI*, (*rule Sup-upper*, *force*)+)

by (*rule conjI*, (*rule Sup-least*, *force*)+)

interpretation *prod*: proto-near-quantale Inf-prod Sup-prod inf-prod less-eq-prod less-prod sup-prod bot-prod
top-prod :: ('a::proto-near-quantale × 'b::proto-near-quantale) *times-prod'*

apply (*standard*, *simp add*: *times-prod'-def* *Sup-prod-def*)

by (*rule conjI*, (*simp add*: *Sup-distr*, *clarify*, *metis*)+)

interpretation *prod*: proto-quantale Inf-prod Sup-prod inf-prod less-eq-prod less-prod sup-prod bot-prod
top-prod :: ('a::proto-quantale × 'b::proto-quantale) *times-prod'*

apply (*standard*, *simp add*: *times-prod'-def* *Sup-prod-def* *less-eq-prod-def*)

by (*rule conjI*, (*simp add*: *Sup-distl*, *metis*)+)

interpretation *prod*: unital-quantale *one-prod times-prod'* Inf-prod Sup-prod inf-prod less-eq-prod less-prod
sup-prod bot-prod top-prod :: ('a::unital-quantale × 'b::unital-quantale)

by *standard* (*simp-all add*: *one-prod-def times-prod'-def mult.assoc*)

4.6 Quantale Modules and Semidirect Products

Quantale modules are extensions of semigroup actions in that a quantale acts on a complete lattice.

locale *unital-quantale-module* =

fixes *act* :: 'a::unital-quantale ⇒ 'b::complete-lattice ⇒ 'b (α)

assumes *act1*: α (x · y) p = α x (α y p)

and *act2* [*simp*]: α 1 p = p

and *act3*: α (⊔ X) p = ⊔ {α x p | x. x ∈ X}

and *act4*: α x (⊔ P) = ⊔ {α x p | p. p ∈ P}

context *unital-quantale-module*

begin

Actions are morphisms. The curried notation is particularly convenient for this.

lemma *act-morph1*: α (x · y) = (α x) ○ (α y)

by (*simp add*: *fun-eq-iff act1*)

lemma *act-morph2*: α 1 = *id*

by (*simp add*: *fun-eq-iff*)

lemma *emp-act*: α (⊔ {}) p = ⊥

by (*simp only*: *act3*, *force*)

lemma *emp-act-var*: α ⊥ p = ⊥

using *emp-act* **by** *auto*

lemma *act-emp*: α x (⊔ {}) = ⊥

by (*simp only: act4, force*)

lemma *act-emp-var*: $\alpha x \perp = \perp$
 using *act-emp* by *auto*

lemma *act-sup-distl*: $\alpha x (p \sqcup q) = (\alpha x p) \sqcup (\alpha x q)$

proof–

have $\alpha x (p \sqcup q) = \alpha x (\bigsqcup \{p, q\})$

by *simp*

also have $\dots = \bigsqcup \{\alpha x y \mid y. y \in \{p, q\}\}$

by (*rule act4*)

also have $\dots = \bigsqcup \{v. v = \alpha x p \vee v = \alpha x q\}$

by (*metis empty-iff insert-iff*)

finally show *?thesis*

by (*metis Collect-disj-eq Sup-insert ccpo-Sup-singleton insert-def singleton-conv*)

qed

lemma *act-sup-distr*: $\alpha (x \sqcup y) p = (\alpha x p) \sqcup (\alpha y p)$

proof–

have $\alpha (x \sqcup y) p = \alpha (\bigsqcup \{x, y\}) p$

by *simp*

also have $\dots = \bigsqcup \{\alpha v p \mid v. v \in \{x, y\}\}$

by (*rule act3*)

also have $\dots = \bigsqcup \{v. v = \alpha x p \vee v = \alpha y p\}$

by (*metis empty-iff insert-iff*)

finally show *?thesis*

by (*metis Collect-disj-eq Sup-insert ccpo-Sup-singleton insert-def singleton-conv*)

qed

lemma *act-sup-distr-var*: $\alpha (x \sqcup y) = (\alpha x) \sqcup (\alpha y)$

by (*simp add: fun-eq-iff act-sup-distr*)

Next we define the semidirect product of a unital quantale and a complete lattice.

definition *sd-prod* $x y = (fst x \cdot fst y, snd x \sqcup \alpha (fst x) (snd y))$

lemma *sd-distr-aux*:

$\bigsqcup \{snd x \mid x. x \in X\} \sqcup \bigsqcup \{\alpha (fst x) p \mid x. x \in X\} = \bigsqcup \{snd x \sqcup \alpha (fst x) p \mid x. x \in X\}$

proof (*rule antisym, rule sup-least*)

show $\bigsqcup \{snd x \mid x. x \in X\} \leq \bigsqcup \{snd x \sqcup \alpha (fst x) p \mid x. x \in X\}$

proof (*rule Sup-least*)

fix $x :: 'b$

assume $x \in \{snd x \mid x. x \in X\}$

hence $\exists b pa. x \sqcup b = snd pa \sqcup \alpha (fst pa) p \wedge pa \in X$

by *blast*

hence $\exists b. x \sqcup b \in \{snd pa \sqcup \alpha (fst pa) p \mid pa. pa \in X\}$

by *blast*

thus $x \leq \bigsqcup \{snd pa \sqcup \alpha (fst pa) p \mid pa. pa \in X\}$

by (*meson Sup-upper sup.bounded-iff*)

qed

next

show $\bigsqcup \{\alpha (fst x) p \mid x. x \in X\} \leq \bigsqcup \{snd x \sqcup \alpha (fst x) p \mid x. x \in X\}$

proof (*rule Sup-least*)

fix $x :: 'b$

assume $x \in \{\alpha (fst x) p \mid x. x \in X\}$

then have $\exists b pa. b \sqcup x = snd pa \sqcup \alpha (fst pa) p \wedge pa \in X$

by *blast*
then have $\exists b. b \sqcup x \in \{snd\ pa \sqcup \alpha\ (fst\ pa)\ p \mid pa. pa \in X\}$
 by *blast*
then show $x \leq \bigsqcup \{snd\ pa \sqcup \alpha\ (fst\ pa)\ p \mid pa. pa \in X\}$
 by (*meson Sup-upper le-supE*)
qed
next
show $\bigsqcup \{snd\ x \sqcup \alpha\ (fst\ x)\ p \mid x. x \in X\} \leq \bigsqcup \{snd\ x \mid x. x \in X\} \sqcup \bigsqcup \{\alpha\ (fst\ x)\ p \mid x. x \in X\}$
apply (*rule Sup-least*)
apply *safe*
apply (*rule sup-least*)
apply (*metis (mono-tags, lifting) Sup-upper mem-Collect-eq sup.coboundedI1*)
by (*metis (mono-tags, lifting) Sup-upper mem-Collect-eq sup.coboundedI2*)
qed
lemma *sd-distr*: $sd\text{-prod}\ (Sup\text{-prod}\ X)\ y = Sup\text{-prod}\ \{sd\text{-prod}\ x\ y \mid x. x \in X\}$
proof –
have $sd\text{-prod}\ (Sup\text{-prod}\ X)\ y = sd\text{-prod}\ (\bigsqcup \{fst\ x \mid x. x \in X\}, \bigsqcup \{snd\ x \mid x. x \in X\})\ y$
by (*simp add: Sup-prod-def*)
also have
 $\dots = ((\bigsqcup \{fst\ x \mid x. x \in X\}) \cdot fst\ y, (\bigsqcup \{snd\ x \mid x. x \in X\}) \sqcup (\alpha\ (\bigsqcup \{fst\ x \mid x. x \in X\})\ (snd\ y)))$
by (*simp add: sd-prod-def*)
also have
 $\dots = (\bigsqcup \{fst\ x \cdot fst\ y \mid x. x \in X\}, (\bigsqcup \{snd\ x \mid x. x \in X\}) \sqcup (\alpha\ (\bigsqcup \{fst\ x \mid x. x \in X\})\ (snd\ y)))$
by (*simp add: Sup-distr*)
also have
 $\dots = (\bigsqcup \{fst\ x \cdot fst\ y \mid x. x \in X\}, (\bigsqcup \{snd\ x \mid x. x \in X\}) \sqcup (\bigsqcup \{\alpha\ (fst\ x)\ (snd\ y) \mid x. x \in X\}))$
by (*simp add: act3*)
also have $\dots = (\bigsqcup \{fst\ x \cdot fst\ y \mid x. x \in X\}, \bigsqcup \{snd\ x \sqcup (\alpha\ (fst\ x)\ (snd\ y)) \mid x. x \in X\})$
by (*simp only: sd-distr-aur*)
also have $\dots = Sup\text{-prod}\ \{(fst\ x \cdot fst\ y, snd\ x \sqcup (\alpha\ (fst\ x)\ (snd\ y))) \mid x. x \in X\}$
by (*simp add: Sup-prod-def, metis*)
finally show *?thesis*
by (*simp add: sd-prod-def*)
qed
lemma *sd-distl-aur*: $Y \neq \{\}\ \Longrightarrow\ p \sqcup (\bigsqcup \{\alpha\ x\ (snd\ y) \mid y. y \in Y\}) = \bigsqcup \{p \sqcup \alpha\ x\ (snd\ y) \mid y. y \in Y\}$
proof (*rule antisym, rule sup-least*)
show $Y \neq \{\}\ \Longrightarrow\ p \leq \bigsqcup \{p \sqcup \alpha\ x\ (snd\ y) \mid y. y \in Y\}$
proof –
assume $Y \neq \{\}$
hence $\exists b. b \in \{p \sqcup \alpha\ x\ (snd\ pa) \mid pa. pa \in Y\} \wedge p \leq b$
by *fastforce*
thus $p \leq \bigsqcup \{p \sqcup \alpha\ x\ (snd\ pa) \mid pa. pa \in Y\}$
by (*meson Sup-upper2*)
qed
next
show $Y \neq \{\}\ \Longrightarrow\ \bigsqcup \{\alpha\ x\ (snd\ y) \mid y. y \in Y\} \leq \bigsqcup \{p \sqcup \alpha\ x\ (snd\ y) \mid y. y \in Y\}$
apply (*rule Sup-least*)
proof –
fix $xa :: 'b$
assume $xa \in \{\alpha\ x\ (snd\ y) \mid y. y \in Y\}$
then have $\exists b. (\exists pa. b = p \sqcup \alpha\ x\ (snd\ pa) \wedge pa \in Y) \wedge xa \leq b$
by *fastforce*
then have $\exists b. b \in \{p \sqcup \alpha\ x\ (snd\ pa) \mid pa. pa \in Y\} \wedge xa \leq b$


```

    by blast
  then show  $xa \leq \sqcup \{p \sqcup \alpha x (snd pa) \mid pa. pa \in Y\}$ 
    by (meson Sup-upper2)
qed
next
show  $Y \neq \{\} \implies \sqcup \{p \sqcup \alpha x (snd y) \mid y. y \in Y\} \leq p \sqcup \sqcup \{\alpha x (snd y) \mid y. y \in Y\}$ 
  apply (rule Sup-least)
  apply safe
  by (metis (mono-tags, lifting) Sup-le-iff le-sup-iff mem-Collect-eq sup-ge1 sup-ge2)
qed

```

lemma *sd-distl*: $Y \neq \{\} \implies sd\text{-prod } x (Sup\text{-prod } Y) = Sup\text{-prod } \{sd\text{-prod } x y \mid y. y \in Y\}$

proof –

```

  assume a:  $Y \neq \{\}$ 
  have  $sd\text{-prod } x (Sup\text{-prod } Y) = sd\text{-prod } x (\sqcup \{fst y \mid y. y \in Y\}, \sqcup \{snd y \mid y. y \in Y\})$ 
    by (simp add: Sup-prod-def)
  also have  $\dots = ((fst x) \cdot (\sqcup \{fst y \mid y. y \in Y\}), (snd x \sqcup (\alpha (fst x) (\sqcup \{snd y \mid y. y \in Y\}))))$ 
    by (simp add: sd-prod-def)
  also have  $\dots = (\sqcup \{fst x \cdot fst y \mid y. y \in Y\}, (snd x \sqcup (\alpha (fst x) (\sqcup \{snd y \mid y. y \in Y\}))))$ 
    by (simp add: Sup-distl)
  also have  $\dots = (\sqcup \{fst x \cdot fst y \mid y. y \in Y\}, (snd x \sqcup (\sqcup \{\alpha (fst x) (snd y) \mid y. y \in Y\})))$ 
    by (simp add: act4, meson)
  also have  $\dots = (\sqcup \{fst x \cdot fst y \mid y. y \in Y\}, \sqcup \{snd x \sqcup (\alpha (fst x) (snd y)) \mid y. y \in Y\})$ 
    using a sd-distl-aux by blast
  also have  $\dots = Sup\text{-prod } \{(fst x \cdot fst y, snd x \sqcup (\alpha (fst x) (snd y))) \mid y. y \in Y\}$ 
    by (simp add: Sup-prod-def, metis)
  finally show ?thesis
    by (simp add: sd-prod-def)
qed

```

definition *sd-unit* = $(1, \perp)$

lemma *sd-unitl* [simp]: $sd\text{-prod } sd\text{-unit } x = x$
 by (simp add: sd-prod-def sd-unit-def)

lemma *sd-unitr* [simp]: $sd\text{-prod } x sd\text{-unit} = x$
 apply (simp add: sd-prod-def sd-unit-def)
 using act-emp by force

The following counterexamples rule out that semidirect products of quantales and complete lattices form quantales. The reason is that the right annihilation law fails.

lemma $sd\text{-prod } x (Sup\text{-prod } Y) = Sup\text{-prod } \{sd\text{-prod } x y \mid y. y \in Y\}$
 oops

lemma $sd\text{-prod } x bot\text{-prod} = bot\text{-prod}$
 oops

However we can show that semidirect products of (unital) quantales with complete lattices form weak (unital) quantales.

interpretation *dp-quantale*: *weak-quantale sd-prod Inf-prod Sup-prod inf-prod less-eq-prod less-prod sup-prod bot-prod top-prod*
 apply standard
 apply (simp-all add: sd-distl sd-distr)
 apply (simp-all add: sd-prod-def Inf-prod-def Sup-prod-def bot-prod-def sup-prod-def top-prod-def inf-prod-def less-eq-prod-def less-prod-def)

by (rule conjI, simp add: mult.assoc, simp add: act1 act-sup-distl sup-assoc)

interpretation *dpu-quantale*: unital-weak-quantale sd-unit sd-prod Inf-prod Sup-prod inf-prod less-eq-prod less-prod sup-prod bot-prod top-prod

by (standard; simp-all)

end

end

5 Binary Modalities and Relational Convolution

theory *Binary-Modalities*

imports *Quantales*

begin

5.1 Auxiliary Properties

lemma *SUP-is-Sup*: $(SUP f \in F. f y) = \bigsqcup \{(f::'a \Rightarrow 'b::proto-near-quantale) y \mid f. f \in F\}$

proof (rule antisym)

fix $f::'a \Rightarrow 'b::proto-near-quantale$

have $f \in F \Longrightarrow f y \in \{f y \mid f. f \in F\}$

by (simp add: Setcompr-eq-image)

hence $f \in F \Longrightarrow f y \leq \bigsqcup \{f y \mid f. f \in F\}$

by (simp add: Sup-upper)

thus $(SUP f \in F. f y) \leq \bigsqcup \{(f::'a \Rightarrow 'b::proto-near-quantale) y \mid f. f \in F\}$

by (simp add: Setcompr-eq-image)

next

fix x

have $x \in \{f y \mid f. f \in F\} \Longrightarrow x \leq (SUP f \in F. f y)$

using *mk-disjoint-insert* by force

thus $Sup \{(f::'a \Rightarrow 'b::proto-near-quantale) y \mid f. f \in F\} \leq (SUP f \in F. f y)$

by (simp add: Setcompr-eq-image)

qed

lemma *bmod-auxl*: $\{x \cdot g z \mid x. \exists f. x = f y \wedge f \in F\} = \{f y \cdot g z \mid f. f \in F\}$

by force

lemma *bmod-auxr*: $\{f y \cdot x \mid x. \exists g. x = g z \wedge g \in G\} = \{f y \cdot g z \mid g. g \in G\}$

by force

lemma *bmod-assoc-aux1*:

$\bigsqcup \{ \bigsqcup \{(f::'a \Rightarrow 'b::proto-near-quantale) u \cdot g v \cdot h w \mid u v. R y u v\} \mid y w. R x y w\}$
 $= \bigsqcup \{(f u \cdot g v) \cdot h w \mid u v y w. R y u v \wedge R x y w\}$

apply (rule antisym)

apply (rule Sup-least, safe)

apply (intro Sup-least Sup-upper, force)

apply (rule Sup-least, safe)

by (rule Sup-upper2, auto)+

lemma *bmod-assoc-aux2*:

$\bigsqcup \{ \bigsqcup \{(f::'a \Rightarrow 'b::proto-near-quantale) u \cdot g v \cdot h w \mid v w. R y v w\} \mid u y. R x u y\}$
 $= \bigsqcup \{f u \cdot g v \cdot h w \mid u v w y. R y v w \wedge R x u y\}$

apply (rule antisym)

apply (*rule Sup-least, safe*)
apply (*intro Sup-least Sup-upper, force*)
apply (*rule Sup-least, safe*)
by (*rule Sup-upper2, auto*)+

5.2 Binary Modalities

Most of the development in the papers mentioned in the introduction generalises to proto-near-quantales. Binary modalities are interesting for various substructural logics over ternary Kripke frames. They also arise, e.g., as chop modalities in interval logics or as separation conjunction in separation logic. Binary modalities can be understood as a convolution operation parametrised by a ternary operation. Our development yields a unifying framework.

We would prefer a notation that is more similar to our articles, that is, $f *_R g$, but we don't know how we could index an infix operator by a variable in Isabelle.

definition *bmod-comp* :: $('a \Rightarrow 'b \Rightarrow 'c \Rightarrow \text{bool}) \Rightarrow ('b \Rightarrow 'd::\text{proto-near-quantale}) \Rightarrow ('c \Rightarrow 'd) \Rightarrow 'a \Rightarrow 'd$ (\otimes) **where**

$$\otimes R f g x = \bigsqcup \{f y \cdot g z \mid y z. R x y z\}$$

definition *bmod-bres* :: $('c \Rightarrow 'b \Rightarrow 'a \Rightarrow \text{bool}) \Rightarrow ('b \Rightarrow 'd::\text{proto-near-quantale}) \Rightarrow ('c \Rightarrow 'd) \Rightarrow 'a \Rightarrow 'd$ (\triangleleft) **where**

$$\triangleleft R f g x = \prod \{(f y) \rightarrow (g z) \mid y z. R z y x\}$$

definition *bmod-fres* :: $('b \Rightarrow 'a \Rightarrow 'c \Rightarrow \text{bool}) \Rightarrow ('b \Rightarrow 'd::\text{proto-near-quantale}) \Rightarrow ('c \Rightarrow 'd) \Rightarrow 'a \Rightarrow 'd$ (\triangleright) **where**

$$\triangleright R f g x = \prod \{(f y) \leftarrow (g z) \mid y z. R y x z\}$$

lemma *bmod-un-rel*: $\otimes (R \sqcup S) = \otimes R \sqcup \otimes S$

apply (*clarsimp simp: fun-eq-iff bmod-comp-def Sup-union-distrib[symmetric] Collect-disj-eq[symmetric]*)
by (*metis (no-types, lifting)*)

lemma *bmod-Un-rel*: $\otimes (\bigsqcup \mathcal{R}) f g x = \bigsqcup \{\otimes R f g x \mid R. R \in \mathcal{R}\}$

apply (*simp add: bmod-comp-def*)

apply (*rule antisym*)

apply (*rule Sup-least, safe*)

apply (*rule Sup-upper2, force*)

apply (*rule Sup-upper, force*)

apply (*rule Sup-least, safe*)+

by (*metis (mono-tags, lifting) Sup-upper mem-Collect-eq*)

lemma *bmod-sup-fun1*: $\otimes R (f \sqcup g) = \otimes R f \sqcup \otimes R g$

apply (*clarsimp simp add: fun-eq-iff bmod-comp-def sup-distr*)

apply (*rule antisym*)

apply (*intro Sup-least, safe*)

apply (*rule sup-least*)

apply (*intro le-supI1 Sup-upper, force*)

apply (*intro le-supI2 Sup-upper, force*)

apply (*rule sup-least*)

by (*intro Sup-least, safe, rule Sup-upper2, force, simp*)+

lemma *bmod-Sup-fun1*: $\otimes R (\bigsqcup \mathcal{F}) g x = \bigsqcup \{\otimes R f g x \mid f. f \in \mathcal{F}\}$

proof –

have $\otimes R (\bigsqcup \{f. f \in \mathcal{F}\}) g x = \bigsqcup \{\bigsqcup \{f y \mid f. f \in \mathcal{F}\} \cdot g z \mid y z. R x y z\}$

by (*simp add: bmod-comp-def SUP-is-Sup*)

also have $\dots = \bigsqcup \{\bigsqcup \{f y \cdot g z \mid f. f \in \mathcal{F}\} \mid y z. R x y z\}$

by (*simp add: Sup-distr bmod-auxl*)
 also have ... = $\sqcup \{ \sqcup \{ f y \cdot g z \mid y z. R x y z \} \mid f. f \in \mathcal{F} \}$
 apply (*rule antisym*)
 by ((*rule Sup-least, safe*)⁺, (*rule Sup-upper2, force, rule Sup-upper, force*)⁺)
 finally show *?thesis*
 by (*simp add: bmod-comp-def*)
 qed

lemma *bmod-sup-fun2*: $\otimes R (f::'a \Rightarrow 'b::\text{weak-proto-quantale}) (g \sqcup h) = \otimes R f g \sqcup \otimes R f h$
 apply (*clarsimp simp add: fun-eq-iff bmod-comp-def sup-distl*)
 apply (*rule antisym*)
 apply (*intro Sup-least, safe*)
 apply (*rule sup-least*)
 apply (*intro le-supI1 Sup-upper, force*)
 apply (*intro le-supI2 Sup-upper, force*)
 apply (*rule sup-least*)
 by (*intro Sup-least, safe, rule Sup-upper2, force, simp*)⁺

lemma *bmod-Sup-fun2-weak*:
 assumes $\mathcal{G} \neq \{ \}$
 shows $\otimes R f (\sqcup \mathcal{G}) x = \sqcup \{ \otimes R f (g::'a \Rightarrow 'b::\text{weak-proto-quantale}) x \mid g. g \in \mathcal{G} \}$
proof –
 have *set*: $\bigwedge z. \{ g z \mid g::'a \Rightarrow 'b. g \in \mathcal{G} \} \neq \{ \}$
 using *assms* by *blast*
 have $\otimes R f (\sqcup \{ g. g \in \mathcal{G} \}) x = \sqcup \{ f y \cdot \sqcup \{ g z \mid g. g \in \mathcal{G} \} \mid y z. R x y z \}$
 by (*simp add: bmod-comp-def SUP-is-Sup*)
 also have ... = $\sqcup \{ \sqcup \{ f y \cdot g z \mid g. g \in \mathcal{G} \} \mid y z. R x y z \}$
 by (*simp add: weak-Sup-distl[OF set] bmod-auxr*)
 also have ... = $\sqcup \{ \sqcup \{ f y \cdot g z \mid y z. R x y z \} \mid g. g \in \mathcal{G} \}$
 apply (*rule antisym*)
 by ((*rule Sup-least, safe*)⁺, (*rule Sup-upper2, force, rule Sup-upper, force*)⁺)
 finally show *?thesis*
 by (*auto simp: bmod-comp-def*)
 qed

lemma *bmod-Sup-fun2*: $\otimes R f (\sqcup \mathcal{G}) x = \sqcup \{ \otimes R f (g::'a \Rightarrow 'b::\text{proto-quantale}) x \mid g. g \in \mathcal{G} \}$
proof –
 have $\otimes R f (\sqcup \{ g. g \in \mathcal{G} \}) x = \sqcup \{ f y \cdot \sqcup \{ g z \mid g. g \in \mathcal{G} \} \mid y z. R x y z \}$
 by (*simp add: bmod-comp-def SUP-is-Sup*)
 also have ... = $\sqcup \{ \sqcup \{ f y \cdot g z \mid g. g \in \mathcal{G} \} \mid y z. R x y z \}$
 by (*simp add: Sup-distl bmod-auxr*)
 also have ... = $\sqcup \{ \sqcup \{ f y \cdot g z \mid y z. R x y z \} \mid g. g \in \mathcal{G} \}$
 apply (*rule antisym*)
 by ((*rule Sup-least, safe*)⁺, (*rule Sup-upper2, force, rule Sup-upper, force*)⁺)
 finally show *?thesis*
 by (*auto simp: bmod-comp-def*)
 qed

lemma *bmod-comp-bres-galois*: $(\forall x. \otimes R f g x \leq h x) \longleftrightarrow (\forall x. g x \leq \triangleleft R f h x)$
 oops

The following Galois connection requires functions into proto-quantales.

lemma *bmod-comp-bres-galois*: $(\forall x. \otimes R (f::'a \Rightarrow 'b::\text{proto-quantale}) g x \leq h x) \longleftrightarrow (\forall x. g x \leq \triangleleft R f h x)$
proof –

```

have (∀ x. ⊗ R f g x ≤ h x) ←→ (∀ x y z. R x y z → (f y) · (g z) ≤ h x)
  apply (simp add: bmod-comp-def, standard, safe)
  apply (metis (mono-tags, lifting) CollectI Sup-le-iff)
  by (rule Sup-least, force)
also have ... ←→ (∀ x y z. R x y z → g z ≤ (f y) → (h x))
  by (simp add: bres-galois)
finally show ?thesis
  apply (simp add: fun-eq-iff bmod-bres-def)
  apply standard
  using le-Inf-iff apply fastforce
  by (metis (mono-tags, lifting) CollectI le-Inf-iff)
qed

```

lemma *bmod-comp-fres-galois*: $(\forall x. \otimes R f g x \leq h x) \longleftrightarrow (\forall x. f x \leq \triangleright R h g x)$

proof –

```

have (∀ x. ⊗ R f g x ≤ h x) ←→ (∀ x y z. R x y z → (f y) · (g z) ≤ h x)
  apply (simp add: bmod-comp-def, standard, safe)
  apply (metis (mono-tags, lifting) CollectI Sup-le-iff)
  by (rule Sup-least, force)
also have ... ←→ (∀ x y z. R x y z → f y ≤ (h x) ← (g z))
  by (simp add: fres-galois)
finally show ?thesis
  apply (simp add: bmod-fres-def fun-eq-iff)
  apply standard
  using le-Inf-iff apply fastforce
  by (metis (mono-tags, lifting) CollectI le-Inf-iff)
qed

```

5.3 Relational Convolution and Correspondence Theory

We now fix a ternary relation ρ and can then hide the parameter in a convolution-style notation.

```

class rel-magma =
  fixes ρ :: 'a ⇒ 'a ⇒ 'a ⇒ bool

```

begin

definition *times-rel-fun* :: $('a \Rightarrow 'b :: \text{proto-near-quantale}) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b$ (**infix** \star 70) **where**
 $f \star g = \otimes \rho f g$

lemma *rel-fun-Sup-distl-weak*:

$G \neq \{\}$ $\implies (f :: 'a \Rightarrow 'b :: \text{weak-proto-quantale}) \star \bigsqcup G = \bigsqcup \{f \star g \mid g. g \in G\}$

proof –

```

fix f :: 'a ⇒ 'b and G :: ('a ⇒ 'b) set
assume a: G ≠ {}
show f ⋆ ⓐ G = ⓐ {f ⋆ g | g. g ∈ G}
  apply (clarsimp simp: fun-eq-iff times-rel-fun-def bmod-Sup-fun2-weak[OF a])
  apply (rule antisym)
  apply (rule Sup-least, safe)
  apply (rule SUP-upper2, force+)
  apply (rule SUP-least, safe)
  by (rule Sup-upper2, force+)

```

qed

lemma *rel-fun-Sup-distl*: $(f :: 'a \Rightarrow 'b :: \text{proto-quantale}) \star \bigsqcup G = \bigsqcup \{f \star g \mid g. g \in G\}$
apply (*clarsimp simp: fun-eq-iff times-rel-fun-def bmod-Sup-fun2*)

```

apply (rule antisym)
apply (rule Sup-least, safe)
apply (rule SUP-upper2, force+)
apply (rule SUP-least, safe)
by (rule Sup-upper2, force+)

```

```

lemma rel-fun-Sup-distr:  $\sqcup G \star (f :: 'a \Rightarrow 'b :: \text{proto-near-quantale}) = \sqcup \{g \star f \mid g. g \in G\}$ 
apply (clarsimp simp: fun-eq-iff times-rel-fun-def bmod-Sup-fun1)
apply (rule antisym)
apply (rule Sup-least, safe)
apply (rule SUP-upper2, force+)
apply (rule SUP-least, safe)
by (rule Sup-upper2, force+)

```

end

```

class rel-semigroup = rel-magma +
assumes rel-assoc:  $(\exists y. \varrho y u v \wedge \varrho x y w) \longleftrightarrow (\exists z. \varrho z v w \wedge \varrho x u z)$ 

```

begin

Nitpick produces counterexamples even for weak quantales. Hence one cannot generally lift functions into weak quantales to weak quantales.

```

lemma bmod-assoc:  $\otimes \varrho (\otimes \varrho (f :: 'a \Rightarrow 'b :: \text{weak-quantale}) g) h x = \otimes \varrho f (\otimes \varrho g h) x$ 

```

oops

```

lemma bmod-assoc:  $\otimes \varrho (\otimes \varrho (f :: 'a \Rightarrow 'b :: \text{quantale}) g) h x = \otimes \varrho f (\otimes \varrho g h) x$ 

```

proof –

```

have  $\otimes \varrho (\otimes \varrho f g) h x = \sqcup \{ \sqcup \{ f u \cdot g v \cdot h z \mid u v. \varrho y u v \} \mid y z. \varrho x y z \}$ 
apply (simp add: bmod-comp-def Sup-distr)
apply (rule antisym)
by (intro Sup-least Sup-upper, safe, intro exI conjI, simp-all, rule-tac f = Sup and g = Sup in cong,
auto) +
also have  $\dots = \sqcup \{ f u \cdot g v \cdot h z \mid u v y z. \varrho y u v \wedge \varrho x y z \}$ 
by (simp add: bmod-assoc-aux1)
also have  $\dots = \sqcup \{ f u \cdot g v \cdot h z \mid u v z y. \varrho y v z \wedge \varrho x u y \}$ 
apply (rule antisym)
apply (rule Sup-least, rule Sup-upper, safe)
using rel-assoc apply force
apply (rule Sup-least, rule Sup-upper, safe)
using rel-assoc by blast
also have  $\dots = \sqcup \{ \sqcup \{ f u \cdot g v \cdot h z \mid v z. \varrho y v z \} \mid u y. \varrho x u y \}$ 
by (simp add: bmod-assoc-aux2)
also have  $\dots = \sqcup \{ f u \cdot \sqcup \{ g v \cdot h z \mid v z. \varrho y v z \} \mid u y. \varrho x u y \}$ 
apply (simp add: Sup-distl mult.assoc)
apply (rule antisym)
by (intro Sup-least Sup-upper, safe, intro exI conjI, simp-all, rule-tac f = Sup and g = Sup in cong,
auto) +
finally show ?thesis
by (auto simp: bmod-comp-def)
qed

```

```

lemma rel-fun-assoc:  $((f :: 'a \Rightarrow 'b :: \text{quantale}) \star g) \star h = f \star (g \star h)$ 
by (simp add: times-rel-fun-def fun-eq-iff bmod-assoc[symmetric])

```

end

lemma $\otimes R (\otimes R f f) f x = \otimes R f (\otimes R f f) x$

oops

```
class rel-monoid = rel-semigroup +
  fixes  $\xi :: 'a \text{ set}$ 
  assumes unittl-ex:  $\exists e \in \xi. \varrho x e x$ 
  and unitr-ex:  $\exists e \in \xi. \varrho x x e$ 
  and unittl-eq:  $e \in \xi \implies \varrho x e y \implies x = y$ 
  and unitr-eq:  $e \in \xi \implies \varrho x y e \implies x = y$ 
```

begin

```
lemma xi-prop:  $e1 \in \xi \implies e2 \in \xi \implies e1 \neq e2 \implies \neg \varrho x e1 e2 \wedge \neg \varrho x e2 e1$ 
  using unittl-eq unitr-eq by blast
```

```
definition pid :: 'a  $\Rightarrow$  'b::unital-weak-quantale ( $\delta$ ) where
   $\delta x = (\text{if } x \in \xi \text{ then } 1 \text{ else } \perp)$ 
```

Due to the absence of right annihilation, the right unit law fails for functions into weak quantales.

```
lemma bmod-onel:  $\otimes \varrho f (\delta::'a \Rightarrow 'b::unital-weak-quantale) x = f x$ 
```

oops

A unital quantale is required for this lifting.

```
lemma bmod-onel:  $\otimes \varrho f (\delta::'a \Rightarrow 'b::unital-quantale) x = f x$ 
  apply (simp add: bmod-comp-def pid-def)
  apply (rule antisym)
  apply (rule Sup-least, safe)
  apply (simp add: bres-galois)
  using unitr-eq apply fastforce
  apply (metis bot.extremum)
  by (metis (mono-tags, lifting) Sup-upper mem-Collect-eq unitr-ex)
```

```
lemma bmod-oner:  $\otimes \varrho \delta f x = f x$ 
  apply (simp add: bmod-comp-def pid-def)
  apply (rule antisym)
  apply (rule Sup-least, safe)
  apply (simp add: fres-galois)
  using unittl-eq apply fastforce
  apply (metis bot.extremum)
  by (metis (mono-tags, lifting) Sup-upper mem-Collect-eq unittl-ex)
```

```
lemma pid-unittl [simp]:  $\delta \star f = f$ 
  by (simp add: fun-eq-iff times-rel-fun-def bmod-oner)
```

```
lemma pid-unitr [simp]:  $f \star (\delta::'a \Rightarrow 'b::unital-quantale) = f$ 
  by (simp add: fun-eq-iff times-rel-fun-def bmod-onel)
```

```
lemma bmod-assoc-weak-aux:
   $f u \cdot \bigsqcup \{g v \cdot h z \mid v z. \varrho y v z\} = \bigsqcup \{(f::'a \Rightarrow 'b::weak-quantale) u \cdot g v \cdot h z \mid v z. \varrho y v z\}$ 
  apply (subst weak-Sup-distl)
```

```

using unittl-ex apply force
apply simp
by (metis (no-types, lifting) mult.assoc)

lemma bmod-assoc-weak:  $\otimes \varrho (\otimes \varrho (f :: 'a \Rightarrow 'b :: \text{weak-quantale}) g) h x = \otimes \varrho f (\otimes \varrho g h) x$ 
proof -
  have  $\otimes \varrho (\otimes \varrho f g) h x = \bigsqcup \{ \bigsqcup \{ f u \cdot g v \cdot h z \mid u v. \varrho y u v \} \mid y z. \varrho x y z \}$ 
  apply (simp add: bmod-comp-def Sup-distr)
  apply (rule antisym)
  by (intro Sup-least Sup-upper, safe, intro exI conjI, simp-all, rule-tac f = Sup and g = Sup in cong,
auto)+
  also have  $\dots = \bigsqcup \{ f u \cdot g v \cdot h z \mid u v y z. \varrho y u v \wedge \varrho x y z \}$ 
  by (simp add: bmod-assoc-aux1)
  also have  $\dots = \bigsqcup \{ f u \cdot g v \cdot h z \mid u v z y. \varrho y v z \wedge \varrho x u y \}$ 
  apply (rule antisym)
  apply (rule Sup-least, rule Sup-upper, safe)
  using rel-assoc apply force
  apply (rule Sup-least, rule Sup-upper, safe)
  using rel-assoc by blast
  also have  $\dots = \bigsqcup \{ \bigsqcup \{ f u \cdot g v \cdot h z \mid v z. \varrho y v z \} \mid u y. \varrho x u y \}$ 
  by (simp add: bmod-assoc-aux2)
  also have  $\dots = \bigsqcup \{ f u \cdot \bigsqcup \{ g v \cdot h z \mid v z. \varrho y v z \} \mid u y. \varrho x u y \}$ 
  by (simp add: bmod-assoc-weak-aux)
  finally show ?thesis
  by (auto simp: bmod-comp-def)
qed

lemma rel-fun-assoc-weak:  $((f :: 'a \Rightarrow 'b :: \text{weak-quantale}) \star g) \star h = f \star (g \star h)$ 
by (simp add: times-rel-fun-def fun-eq-iff bmod-assoc-weak[symmetric])

end

lemma (in rel-semigroup)  $\exists id. \forall f x. (\otimes \varrho f id x = f x \wedge \otimes \varrho id f x = f x)$ 

oops

class rel-ab-semigroup = rel-semigroup +
  assumes rel-comm:  $\varrho x y z \Longrightarrow \varrho x z y$ 

begin

lemma bmod-comm:  $\otimes \varrho (f :: 'a \Rightarrow 'b :: \text{ab-quantale}) g = \otimes \varrho g f$ 
by (simp add: fun-eq-iff bmod-comp-def mult commute, meson rel-comm)

lemma  $\otimes \varrho f g = \otimes \varrho g f$ 
oops

lemma bmod-bres-fres-eq:  $\triangleleft \varrho (f :: 'a \Rightarrow 'b :: \text{ab-quantale}) g = \triangleright \varrho g f$ 
by (simp add: fun-eq-iff bmod-bres-def bmod-fres-def bres-fres-eq, meson rel-comm)

lemma rel-fun-comm:  $(f :: 'a \Rightarrow 'b :: \text{ab-quantale}) \star g = g \star f$ 
by (simp add: times-rel-fun-def bmod-comm)

end

```


class *rel-ab-monoid* = *rel-ab-semigroup* + *rel-monoid*

5.4 Lifting to Function Spaces

We lift by interpretation, since we need sort instantiations to be used for functions from PAM's to Quantales. Both instantiations cannot be used in Isabelle at the same time.

interpretation *rel-fun*: *weak-proto-quantale Inf Sup inf less-eq less sup bot top* :: '*a*::*rel-magma* ⇒ '*b*::*weak-proto-quantale times-rel-fun*

by *standard* (*simp-all add: rel-fun-Sup-distr rel-fun-Sup-distl-weak*)

interpretation *rel-fun*: *proto-quantale Inf Sup inf less-eq less sup bot top* :: '*a*::*rel-magma* ⇒ '*b*::*proto-quantale times-rel-fun*

by *standard* (*simp add: rel-fun-Sup-distl*)

Nitpick shows that the lifting of weak quantales to weak quantales does not work for relational semigroups, because associativity fails.

interpretation *rel-fun*: *weak-quantale times-rel-fun Inf Sup inf less-eq less sup bot top*::'*a*::*rel-semigroup* ⇒ '*b*::*weak-quantale*

oops

Associativity is obtained when lifting from relational monoids, but the right unit law doesn't hold in the quantale on the function space, according to our above results. Hence the lifting results into a non-unital quantale, in which only the left unit law holds, as shown above. We don't provide a special class for such quantales. Hence we lift only to non-unital quantales.

interpretation *rel-fun*: *weak-quantale times-rel-fun Inf Sup inf less-eq less sup bot top*::'*a*::*rel-monoid* ⇒ '*b*::*unital-weak-quantale*

by *standard* (*simp-all add: rel-fun-assoc-weak*)

interpretation *rel-fun2*: *quantale times-rel-fun Inf Sup inf less-eq less sup bot top*::'*a*::*rel-semigroup* ⇒ '*b*::*quantale*

by *standard* (*simp add: rel-fun-assoc*)

interpretation *rel-fun2*: *distrib-quantale Inf Sup inf less-eq less sup bot top*::'*a*::*rel-semigroup* ⇒ '*b*::*distrib-quantale times-rel-fun* ..

interpretation *rel-fun2*: *bool-quantale minus uminus inf less-eq less sup bot <top::'a::rel-semigroup ⇒ 'b::bool-quantale> Inf Sup times-rel-fun* ..

interpretation *rel-fun2*: *unital-quantale pid times-rel-fun Inf Sup inf less-eq less sup bot top*::'*a*::*rel-monoid* ⇒ '*b*::*unital-quantale*

by (*standard; simp*)

interpretation *rel-fun2*: *distrib-unital-quantale Inf Sup inf less-eq less sup bot top*::'*a*::*rel-monoid* ⇒ '*b*::*distrib-unital-quantale pid times-rel-fun* ..

interpretation *rel-fun2*: *bool-unital-quantale minus uminus inf less-eq less sup bot <top::'a::rel-monoid ⇒ 'b::bool-unital-quantale> Inf Sup pid times-rel-fun* ..

interpretation *rel-fun*: *ab-quantale times-rel-fun Inf Sup inf less-eq less sup bot top*::'*a*::*rel-ab-semigroup* ⇒ '*b*::*ab-quantale*

by *standard* (*simp add: rel-fun-comm*)

interpretation *rel-fun*: *ab-unital-quantale times-rel-fun Inf Sup inf less-eq less sup bot top::'a::rel-ab-monoid*
 $\Rightarrow 'b::ab-unital-quantale\ pid \ ..$

interpretation *rel-fun2*: *distrib-ab-unital-quantale Inf Sup inf less-eq less sup bot top::'a::rel-ab-monoid*
 $\Rightarrow 'b::distrib-ab-unital-quantale\ times-rel-fun\ pid \ ..$

interpretation *rel-fun2*: *bool-ab-unital-quantale times-rel-fun Inf Sup inf less-eq less sup bot top::'a::rel-ab-monoid*
 $\Rightarrow 'b::bool-ab-unital-quantale\ minus\ uminus\ pid \ ..$

end

6 Unary Modalities

theory *Unary-Modalities*
imports *Binary-Modalities*
begin

Unary modalities arise as specialisations of the binary ones; and as generalisations of the standard (multi-)modal operators from predicates to functions into complete lattices. They are interesting, for instance, in combination with partial semigroups or monoids, for modelling the Halpern-Shoham modalities in interval logics.

6.1 Forward and Backward Diamonds

definition *fdia* :: (*'a* × *'b*) *set* \Rightarrow (*'b* \Rightarrow *'c::complete-lattice*) \Rightarrow *'a* \Rightarrow *'c* ((*|*-) - -) [61,81] 82) **where**
 $(|R| f x) = \sqcup \{f y | y. (x,y) \in R\}$

definition *bdia* :: (*'a* × *'b*) *set* \Rightarrow (*'a* \Rightarrow *'c::complete-lattice*) \Rightarrow *'b* \Rightarrow *'c* ((<-| - -) [61,81] 82) **where**
 $(\langle R | f y) = \sqcup \{f x | x. (x,y) \in R\}$

definition *c1* :: *'a* \Rightarrow *'b::unital-quantale* **where**
 $c1\ x = 1$

The relationship with binary modalities is as follows.

lemma *fdia-bmod-comp*: $(|R| f x) = \otimes (\lambda x\ y\ z. (x,y) \in R) f\ c1\ x$
by (*simp add: fdia-def bmod-comp-def c1-def*)

lemma *bdia-bmod-comp*: $(\langle R | f x) = \otimes (\lambda y\ x\ z. (x,y) \in R) f\ c1\ x$
by (*simp add: bdia-def bmod-comp-def c1-def*)

lemma *bmod-fdia-comp*: $\otimes R\ f\ g\ x = |\{(x,(y,z)) | x\ y\ z. R\ x\ y\ z\}\} (\lambda(x,y). (f\ x) \cdot (g\ y))\ x$
by (*simp add: fdia-def bmod-comp-def*)

lemma *bmod-fdia-comp-var*:
 $\otimes R\ f\ g\ x = |\{(x,(y,z)) | x\ y\ z. R\ x\ y\ z\}\} (\lambda(x,y). (\lambda(v,w).(v \cdot w)) (f\ x, g\ y))\ x$
by (*simp add: fdia-def bmod-comp-def*)

lemma *fdia-im*: $(|R| f x) = \sqcup (f \text{ ' } R \text{ " } \{x\})$
apply (*simp add: fdia-def*)
apply (*rule antisym*)
apply (*intro Sup-least, clarsimp simp: SUP-upper*)
by (*intro SUP-least Sup-upper, force*)

lemma *fdia-un-rel*: $fdia\ (R \cup S) = fdia\ R \sqcup fdia\ S$

apply (*simp add: fun-eq-iff*)
by (*clarsimp simp: fun-eq-iff fdia-im SUP-union Un-Image*)

lemma *fdia-Un-rel*: $(\bigcup \mathcal{R}) f x = \bigsqcup \{|R\rangle f x \mid R. R \in \mathcal{R}\}$

apply (*simp add: fdia-im*)
apply (*rule antisym*)
apply (*intro SUP-least, safe*)
apply (*rule Sup-upper2, force*)
apply (*rule SUP-upper, simp*)
apply (*rule Sup-least*)
by (*clarsimp simp: Image-mono SUP-subset-mono Sup-upper*)

lemma *fdia-sup-fun*: $fdia R (f \sqcup g) = fdia R f \sqcup fdia R g$
by (*simp add: fun-eq-iff fdia-im complete-lattice-class.SUP-sup-distrib*)

lemma *fdia-Sup-fun*: $(\bigsqcup \mathcal{F}) x = \bigsqcup \{|R\rangle f x \mid f. f \in \mathcal{F}\}$

apply (*simp add: fdia-im*)
apply (*rule antisym*)
apply (*rule SUP-least*)
apply (*rule Sup-upper2, force*)
apply (*rule SUP-upper, simp*)
apply (*rule Sup-least, safe*)
apply (*rule SUP-least*)
by (*simp add: SUP-upper2*)

lemma *fdia-seq*: $fdia (R ; S) f x = fdia R (fdia S f) x$
by (*simp add: fdia-im relcomp-Image, metis Image-eq-UN SUP-UNION*)

lemma *fdia-Id* [*simp*]: $(\text{Id}) f x = f x$
by (*simp add: fdia-def*)

6.2 Forward and Backward Boxes

definition *fbox* :: $('a \times 'b) \text{ set} \Rightarrow ('b \Rightarrow 'c::\text{complete-lattice}) \Rightarrow 'a \Rightarrow 'c$ (*[61,81] 82*) **where**
 $(\text{fbox } f) x = \bigcap \{f y \mid y. (x,y) \in R\}$

definition *bbox* :: $('a \times 'b) \text{ set} \Rightarrow ('a \Rightarrow 'c::\text{complete-lattice}) \Rightarrow 'b \Rightarrow 'c$ (*[61,81] 82*) **where**
 $(\text{bbox } f) y = \bigcap \{f x \mid x. (x,y) \in R\}$

6.3 Symmetries and Dualities

lemma *fdia-fbox-demorgan*: $(\text{fbox } f) x = \neg (\text{bbox } (\lambda y. \neg f y)) x$
apply (*simp add: fbox-def fdia-def*)
apply (*rule antisym*)
apply (*rule Sup-least*)
apply (*simp add: Inf-lower compl-le-swap1*)
apply (*simp add: uminus-Inf*)
apply (*rule SUP-least; intro Sup-upper*)
by *auto*

lemma *fbox-fdia-demorgan*: $(\text{fbox } f) x = \neg (\text{bbox } (\lambda y. \neg f y)) x$
apply (*simp add: fbox-def fdia-def*)
apply (*rule antisym*)
apply (*simp add: uminus-Sup*)
apply (*rule INF-greatest; rule Inf-lower*)
apply *auto[1]*

apply (rule *Inf-greatest*)
by (simp add: *Sup-upper compl-le-swap2*)

lemma *bdia-bbox-demorgan*: $(\langle R \mid (f::'b \Rightarrow 'c::\text{complete-boolean-algebra}) x) = - [R] (\lambda y. -f y) x$
apply (simp add: *bbox-def bdia-def*)
apply (rule *antisym*)
apply (rule *Sup-least*)
apply (simp add: *Inf-lower compl-le-swap1*)
apply (simp add: *uminus-Inf*)
apply (rule *SUP-least; intro Sup-upper*)
by *auto*

lemma *bbox-bdia-demorgan*: $([R] (f::'b \Rightarrow 'c::\text{complete-boolean-algebra}) x) = - \langle R \mid (\lambda y. -f y) x$
apply (simp add: *bbox-def bdia-def*)
apply (rule *antisym*)
apply (simp add: *uminus-Sup*)
apply (rule *INF-greatest; rule Inf-lower*)
apply *auto[1]*
apply (rule *Inf-greatest*)
by (simp add: *Sup-upper compl-le-swap2*)

lemma *fdia-bdia-conv*: $(\mid R) f x = \langle \text{converse } R \mid f x$
by (simp add: *fdia-def bdia-def*)

lemma *fbox-bbox-conv*: $(\mid R) f x = [\text{converse } R] f x$
by (simp add: *fbox-def bbox-def*)

lemma *fdia-bbox-galois*: $(\forall x. (\mid R) f x \leq g x) \longleftrightarrow (\forall x. f x \leq [R] g x)$
apply (standard, simp-all add: *fdia-def bbox-def*)
apply *safe*
apply (rule *Inf-greatest*)
apply (force simp: *Sup-le-iff*)
apply (rule *Sup-least*)
by (force simp: *le-Inf-iff*)

lemma *bdia-fbox-galois*: $(\forall x. (\langle R \mid f x) \leq g x) \longleftrightarrow (\forall x. f x \leq \mid R) g x)$
apply (standard, simp-all add: *bdia-def fbox-def*)
apply *safe*
apply (rule *Inf-greatest*)
apply (force simp: *Sup-le-iff*)
apply (rule *Sup-least*)
by (force simp: *le-Inf-iff*)

lemma *dia-conjugate*:
 $(\forall x. (\mid R) (f::'b \Rightarrow 'c::\text{complete-boolean-algebra}) x) \sqcap g x = \perp) \longleftrightarrow (\forall x. f x \sqcap (\langle R \mid g x) = \perp)$
by (simp add: *meet-shunt fdia-bbox-galois bdia-bbox-demorgan*)

lemma *box-conjugate*:
 $(\forall x. (\mid R) (f::'b \Rightarrow 'c::\text{complete-boolean-algebra}) x) \sqcup g x = \top) \longleftrightarrow (\forall x. f x \sqcup ([R] g x) = \top)$

proof –
have $(\forall x. (\mid R) f x \sqcup g x = \top) \longleftrightarrow (\forall x. -g x \leq \mid R) f x)$
by (simp add: *join-shunt sup-commute*)
also have $\dots \longleftrightarrow (\forall x. -g x \leq - [R] (\lambda y. -f y) x)$
by (simp add: *fbox-fdia-demorgan*)
also have $\dots \longleftrightarrow (\forall x. (\mid R) (\lambda y. -f y) x \leq g x)$

```

  by simp
  also have ...  $\longleftrightarrow (\forall x. \neg f x \leq [R] g x)$ 
  by (simp add: fdia-bbox-galois)
  finally show ?thesis
  by (simp add: join-shunt)
qed

end

```

7 Liftings of Partial Semigroups

```

theory Partial-Semigroup-Lifting
  imports Partial-Semigroups Binary-Modalities

```

```
begin
```

First we show that partial semigroups are instances of relational semigroups. Then we extend the lifting results for relational semigroups to partial semigroups.

7.1 Relational Semigroups and Partial Semigroups

Every partial semigroup is a relational partial semigroup.

```

context partial-semigroup
begin

```

```

sublocale rel-partial-semigroup: rel-semigroup R
  by standard (metis add-assoc add-assocD)

```

```
end
```

Every partial monoid is a relational monoid.

```

context partial-monoid
begin

```

```

sublocale rel-partial-monoid: rel-monoid R E
  apply standard
  apply (metis unitl-ex)
  apply (metis unitr-ex)
  apply (metis add-assocD-var1 unitl-ex units-eq-var)
  by (metis add-assocD-var2 unitr-ex units-eq-var)

```

```
end
```

Every PAS is a relational abelian semigroup.

```

context pas
begin

```

```

sublocale rel-pas: rel-ab-semigroup R
  apply standard
  using add-comm by blast

```

```
end
```

Every PAM is a relational abelian monoid.

context *pam*

begin

sublocale *rel-pam*: *rel-ab-monoid* *R E ..*

end

7.2 Liftings of Partial Abelian Semigroups

Functions from partial semigroups into weak quantales form weak proto-quantales.

instantiation *fun* :: (*partial-semigroup*, *weak-quantale*) *weak-proto-quantale*

begin

definition *times-fun* :: ('*a* ⇒ '*b*) ⇒ ('*a* ⇒ '*b*) ⇒ '*a* ⇒ '*b* **where**

times-fun ≡ *rel-partial-semigroup.times-rel-fun*

The following counterexample shows that the associativity law may fail in convolution algebras of functions from partial semigroups into weak quantales.

lemma (*rel-partial-semigroup.times-rel-fun* (*rel-partial-semigroup.times-rel-fun* *f f*) *f*) *x* =
(*rel-partial-semigroup.times-rel-fun* (*f*::'*a*::*partial-semigroup* ⇒ '*b*::*weak-quantale*) (*rel-partial-semigroup.times-rel-fun* *f f*)) *x*

oops

lemma *rel-partial-semigroup.times-rel-fun* (*rel-partial-semigroup.times-rel-fun* *f g*) *h* =

rel-partial-semigroup.times-rel-fun (*f*::'*a*::*partial-semigroup* ⇒ '*b*::*weak-quantale*) (*rel-partial-semigroup.times-rel-fun* *g h*)

oops

instance

by *standard* (*simp-all add: times-fun-def rel-partial-semigroup.rel-fun-Sup-distr rel-magma.rel-fun-Sup-distl-weak*)

end

Functions from partial semigroups into quantales form quantales.

instance *fun* :: (*partial-semigroup*, *quantale*) *quantale*

by *standard* (*simp-all add: times-fun-def rel-partial-semigroup.rel-fun-assoc rel-magma.rel-fun-Sup-distl*)

The following counterexample shows that the right unit law may fail in convolution algebras of functions from partial monoids into weak unital quantales.

lemma (*rel-partial-semigroup.times-rel-fun* (*f*::'*a*::*partial-monoid* ⇒ '*b*::*unital-weak-quantale*) *rel-partial-monoid.pid*)
x = *f x*

oops

Functions from partial monoids into unital quantales form unital quantales.

instantiation *fun* :: (*partial-monoid*, *unital-quantale*) *unital-quantale*

begin

definition *one-fun* :: '*a* ⇒ '*b* **where**

one-fun ≡ *rel-partial-monoid.pid*

instance

by *standard* (*simp-all add: one-fun-def times-fun-def*)

end

These lifting results extend to PASs and PAMs as expected.

instance *fun* :: (*pam*, *ab-quantale*) *ab-quantale*

by *standard* (*simp-all add: times-fun-def rel-pas.rel-fun-comm*)

instance *fun* :: (*pam*, *bool-ab-quantale*) *bool-ab-quantale* ..

instance *fun* :: (*pam*, *bool-ab-unital-quantale*) *bool-ab-unital-quantale* ..

sublocale *ab-quantale* < *abq: pas* (*) λ - . *True*

apply *standard*

apply (*simp-all add: mult-assoc*)

by (*simp add: mult-commute*)

Finally we prove some identities that hold in function spaces.

lemma *times-fun-var*: $(f * g) x = \bigsqcup \{f y * g z \mid y z. R x y z\}$

by (*simp add: times-fun-def rel-partial-semigroup.times-rel-fun-def bmod-comp-def*)

lemma *times-fun-var2*: $(f * g) = (\lambda x. \bigsqcup \{f y * g z \mid y z. R x y z\})$

by (*auto simp: times-fun-var*)

lemma *one-fun-var1* [*simp*]: $x \in E \implies 1 x = 1$

by (*simp add: one-fun-def rel-partial-monoid.pid-def*)

lemma *one-fun-var2* [*simp*]: $x \notin E \implies 1 x = \perp$

by (*simp add: one-fun-def rel-partial-monoid.pid-def*)

lemma *times-fun-canc*: $(f * g) x = \bigsqcup \{f y * g (\text{rquot } x y) \mid y. y \preceq_R x\}$

apply (*rule antisym*)

apply (*simp add: times-fun-var, intro Sup-subset-mono, simp add: Collect-mono-iff*)

using *gR-rel-mult add-canc1* **apply** *force*

apply (*simp add: times-fun-var, intro Sup-subset-mono, simp add: Collect-mono-iff*)

using *gR-rel-defined add-canc2* **by** *fastforce*

lemma *times-fun-prod*: $(f * g) = (\lambda(x, y). \bigsqcup \{f (x, y1) * g (x, y2) \mid y1 y2. R y y1 y2\})$

by (*auto simp: times-fun-var2 times-prod-def D-prod-def*)

lemma *one-fun-prod1* [*simp*]: $y \in E \implies 1 (x, y) = 1$

by (*simp add: E-prod-def*)

lemma *one-fun-prod2* [*simp*]: $y \notin E \implies 1 (x, y) = \perp$

by (*simp add: E-prod-def*)

lemma *fres-galois-funI*: $\forall x. (f * g) x \leq h x \implies f x \leq (h \leftarrow g) x$

by (*meson fres-galois le-funD le-funI*)

lemma *times-fun-prod-canc*: $(f * g) (x, y) = \bigsqcup \{f (x, z) * g (x, \text{rquot } y z) \mid z. z \preceq_R y\}$

apply (*simp add: times-fun-prod*)

by (*metis (no-types, lifting) gR-rel-defined gR-rel-mult add-canc1 add-canc2*)

The following statement shows, in a generalised setting, that the magic wand operator of separation logic can be lifted from the heap subtraction operation generalised to a cancellative PAM.

lemma *fres-lift*: $(fres\ f\ g)\ (x::'b::cancellative-pam) = \sqcap \{(f\ y) \leftarrow (g\ z) \mid y\ z \cdot z \preceq_R y \wedge x = rquot\ y\ z\}$

proof (*rule antisym*)

```

{ fix h y z
  assume assms: h · g ≤ f z ≤R y x = rquot y z
  moreover hence D z x
    using add-rquot by blast
  moreover hence h x · g z ≤ (h · g) (x ⊕ z)
    using add-comm by (auto simp add: times-fun-var intro!: Sup-upper)
  moreover hence (h * g) (x ⊕ z) ≤ f (z ⊕ x)
    by (simp add: ⟨D z x⟩ calculation(1) le-funD add-comm)
  ultimately have h x ≤ (f (z ⊕ x)) ← (g z)
    by (auto simp: fres-def intro: Sup-upper)
  from this and assms have h (rquot y z) ≤ (f y) ← (g z)
    by (simp add: add-canc2)
}
thus (f ← g) x ≤  $\sqcap \{(f\ y) \leftarrow (g\ z) \mid y\ z \cdot z \preceq_R y \wedge x = rquot\ y\ z\}$ 
  by (clarsimp simp: fres-def intro!: Inf-greatest SUP-least)
next
have  $\sqcap \{(f\ y) \leftarrow (g\ z) \mid y\ z \cdot z \preceq_R y \wedge x = rquot\ y\ z\} \leq Sup\{x \cdot x \cdot g \leq f\} x$ 
  apply (clarsimp simp: times-fun-var intro!: SUP-upper le-funI Sup-least)
  apply (simp add: fres-galois)
  apply (intro Inf-lower)
  apply safe
  by (metis gR-rel-mult add-canc1 add-comm)
thus  $\sqcap \{(f\ y) \leftarrow (g\ z) \mid y\ z \cdot z \preceq_R y \wedge x = rquot\ y\ z\} \leq (f \leftarrow g) x$ 
  by (simp add: fres-def)
qed
end

```

References

- [1] B. Dongol, V. B. F. Gomes, and G. Struth. A program construction and verification tool for separation logic. In *MPC 2015*, volume 9129 of *LNCS*, pages 137–158. Springer, 2015.
- [2] B. Dongol, I. J. Hayes, and G. Struth. Convolution as a unifying concept: Applications in separation logic, interval calculi, and concurrency. *ACM TOCL*, 17(3):15, 2016.
- [3] B. Dongol, I. J. Hayes, and G. Struth. Relational convolution, generalised modalities and incidence algebras. *CoRR*, abs/1702.04603, 2017.
- [4] J. Y. Halpern and Y. Shoham. A propositional modal logic of time intervals. *J. ACM*, 38(4):935–962, 1991.
- [5] T. Hoare, B. Möller, G. Struth, and I. Wehrman. Concurrent Kleene algebra and its foundations. *J. Logic and Algebraic Programming*, 80(6):266–296, 2011.
- [6] B. C. Moszkowski. A complete axiomatization of interval temporal logic with infinite time. In *LICS 2000*, pages 241–252. IEEE Computer Society, 2000.
- [7] Y. Venema. A modal logic for chopping intervals. *Journal of Logic and Computation*, 1(4):453–476, 1991.
- [8] C. Zhou and M. R. Hansen. *Duration Calculus: A Formal Approach to Real-Time Systems*. Springer, 2004.