A Mechanised Abstract Formalisation of Concept Lattices

Wolfram Kahl*

McMaster University, Hamilton, Ontario, Canada
kahl@cas.mcmaster.ca

Abstract. Using the dependently-typed programming language Agda, we formalise a category of algebraic contexts with relational homomorphisms presented by [Jip12, Mos13]. We do this in the abstract setting of locally ordered categories with converse (OCCs) with residuals and direct powers, without requiring meets (as in allegories) or joins (as in Kleene categories). The abstract formalisation has the advantage that it can be used both for theoretical reasoning, and for executable implementations, by instantiating it with appropriate choices of concrete OCCs.

1 Introduction

Formal concept analysis (FCA) [Wil05] typically starts from a *context* (E, A, R) consisting of a set E of *entities* (or "objects"), a set A of *attributes*, and an *incidence* relation R from entities to attributes. In such a context, "concepts" arise as "Galois-closed" subsets of E respectively A, and form complete "concept lattices".

In a recent development, M. A. Moshier [Mos13] defined a novel *relational* context homomorphism concept that gives rise to a category of contexts that is dual to the category of complete meet semilattices. This is in contrast with the FCA literature, which typically derives the context homomorphism concept from that used for the concept lattices, as for example in [HKZ06], with the notable exception of Erné, who studied context homomorphisms consisting of pairs of mappings [Ern05].

Jipsen [Jip12] published the central definitions of Moshier's [Mos13] approach, and developed it further to obtain categories of context representations of not only complete lattices, but also different kinds of semirings.

We now set out to mechanise the basis of these developments, and for the sake of reusability we abstract the sets and relations that constitute contexts to objects and morphisms of suitable categories and semigroupoids. Besides the mechanised formalisation itself, our main contribution is the insight that Moshier's relational context category can be formalised in categories of "abstract relations" where neither meet (intersection) nor join (union) are available, and that a large part of this development does not even require the presence of identity relations.

* This research is supported by the National Science and Engineering Research Council of Canada, NSERC.

P. Höfner et al. (Eds.): RAMiCS 2014, LNCS 8428, pp. 242–260, 2014.

Overview

We start with an introduction to essential features of the dependently-typed programming language and proof checker Agda2 (in the following just referred to as Agda) and an overview of our RATH-Agda formalisation of categoric abstractions of functions and relations in Sect. 3.

Since formal concept analysis concentrates on subsets of the constituent sets of the contexts we are interested in, we formalise an abstract version of element relations corresponding to the direct powers of [BSZ86, BSZ89] or the power allegories of [FS90], directly in the setting of locally ordered semigroupoids with converse (OSGCs) in Sect. 4. Adding also residuals to that setting (Sect. 5) proves sufficient for the formalisation of the "compatibility conditions" of Moshier's relational context homomorphisms, in Sect. 6. Defining composition of these homomorphisms requires making identity relations available, that is, moving from semigroupoids to categories; locally ordered categories with converse (OCCs) and residuals and a power operator are sufficient to formalise the context category, in Sect. 7. We conclude with additional discussion of the merits of our abstract formalisation.

The Agda source code from this project, including the modules discussed in this paper, are available on-line at `http://relmics.mcmaster.ca/RATH-Agda/`.

2 Agda Notation

This paper reports on a development in the dependently typed functional programming language and proof assistant Agda [Nor07] based on Martin-Löf type theory. Since Agda has been designed with a clear focus on both readability and writability, we present all mathematical content (except some informal analogies) in Agda notation, as an excerpt of the actual mechanically checked development.

Many Agda features will be explained when they are first used; here we only summarise a few *essential* aspects to make our use of Agda as the mathematical notation in the remainder of this paper more widely accessible.

Syntactically and "culturally", Agda frequently seems quite close to Haskell. However, the syntax of Agda is much more flexible: Almost any sequence of non-space characters is a legal lexeme, permitting the habit of choosing variable names for properties that abbreviate for example their left-hand sides, as for example $\Lambda_{\S\in\breve{}} : \ldots \to \quad \Lambda_0\ \mathsf{R}\ \S\ \in\ \breve{}\ \approx\ \mathsf{R}$ in Sect. 4 below, or names for proof values that reflect their type, e.g., $\mathsf{x}{\approx}\mathsf{y} : \mathsf{x} \approx \mathsf{y}$. Infix operators, and indeed mixfix operators of arbitrary arity, have names that contain underscore characters "_" in the positions of the first explicit arguments; below we use a binary infix operator _≈_ for morphism equality in semigroupoids and categories, and the "circumfix" operator [_] in Sect. 7.

Braces "{...}" in a function type "$\{\mathsf{name} : \mathsf{type}_1\} \to \mathsf{type}_2 \to \ldots$" declare the first argument to be *implicit*; a function (say, "f") with this type can have this implicit argument supplied in three ways:

- "f v_2" supplies the explicit second argument v_2 explicitly, and thereby supplies the name argument implicitly, which requires that the type checker can determine its value uniquely;
- "f $\{v_1\}$ v_2" supplies the name argument explicitly by position, since it is the first implicit argument;
- "f $\{\mathsf{name} = v_1\}$ v_2" supplies the name argument explicitly by name, which is useful when earlier implicit arguments can be inferred by the type checker.

Such implicit argument positions are usually declared for arguments that supply the types to other arguments and can therefore be inferred from the latter; the use of implicit arguments in Agda largely corresponds to general mathematical practice.

Since Agda is strongly normalising and has no undefined values, the underlying semantics is quite different from that of Haskell. In particular, since Agda is dependently typed, it does not have Haskell's distinction between terms, types, and kinds (the "types of the types"). The Agda constant Set_0 corresponds to the Haskell kind *; it is the type of all "normal" datatypes and is at the bottom of the hierarchy of type-theoretic universes in Agda. Universes Set ℓ are distinguished by universe indices for which we use names like ℓ ℓ_1 ℓa ℓi : Level, where Level is an opaque special-purpose variant of the natural numbers; we write its maximum operator as _⊔_. This *universe polymorphism* is essential for being able to talk about both "small" and "large" categories or relation algebras, and we always choose our Level parameters so as to enable maximal reusability of our definitions. (We include full Level information in all our types, although it does not essentially contribute to our development.)

Since types in Agda may be uninhabited, predicates use Set ℓ as result type. For example, we define the predicate isIdentity : {A : Obj} → Mor A A → Set ℓk below so that the application "isIdentity F" denotes the type of proofs that F : Mor A A is an identity morphism, which means that isIdentity F is an inhabited type if and only if F is an identity morphism.

3 Semigroupoids, Categories, OSGCs, OCCs

In [Kah11b], we presented a relatively fine-grained modularisation of sub-theories of division allegories, following our work on using semigroupoids to provide the theory of finite relations between infinite types, as they frequently occur as data structures in programming [Kah08], and on collagories [Kah11a] ("distributive allegories without zero morphisms"). In this section, we present two monolithic definitions that provide appropriate foundations for most of the discussion in this paper. Each of these two definitions bundles a large number of theories of the RATH-Agda libraries summarised in [Kah11b].

We show here a monolithic definition of semigroupoids (i.e., "categories without identity morphisms"), which can be used as an alternative to the one within the fine-grained theory hierarchy of [Kah11b]. We make no provisions for user-defined equality on objects, so Obj is a Set. Morphisms have equality _≈_; for

any two objects A and B we have Hom A B : Setoid j k, with the standard library providing an implementation of the standard type-theoretic concept of *setoid* as a carrier set together with an equivalence relation that is considered as equality on the carrier, much like the equality test == provided by the class Eq in Haskell.[1] In [Kah11b] and in the Agda development [Kah14] underlying the Agda theories for the later sections of this paper, only the Levels and Obj are parameters of the Semigroupoid type; for making the presentation in this section easier to follow, we choose to have also Hom as a parameter of the Semigroupoid′ type.

```
record Semigroupoid′ {ℓi ℓj ℓk : Level} {Obj : Set ℓi}
                     (Hom : Obj → Obj → Setoid ℓj ℓk)
                     : Set (ℓi ⊔ ℓj ⊔ ℓk) where
  Mor : Obj → Obj → Set ℓj
  Mor = λ A B → Setoid.Carrier (Hom A B)
  infix 4 _≈_; infixr 9 _⨟_
  _≈_ = λ {A} {B} → Setoid._≈_ (Hom A B)
  field _⨟_   : {A B C : Obj} → Mor A B → Mor B C → Mor A C
        ⨟-cong : {A B C : Obj} {f₁ f₂ : Mor A B} {g₁ g₂ : Mor B C}
                 → f₁ ≈ f₂ → g₁ ≈ g₂ → f₁ ⨟ g₁ ≈ f₂ ⨟ g₂
        ⨟-assoc : {A B C D : Obj} {f : Mor A B} {g : Mor B C} {h : Mor C D}
                  → (f ⨟ g) ⨟ h ≈ f ⨟ (g ⨟ h)
```

Using the infix declarations above, we in particular make morphism composition _⨟_ have higher precedence than morphism equality, which allows us to mostly follow mathematical parenthesisation conventions. Function application, written as juxtaposition (here only occurring in Mor A B etc.) has higher precedence than any infix/mixfix operator, and associates to the left: "Mor A B" is "(Mor A) B", while the **infixr** declaration above specifies that morphism composition associates to the right: "f ⨟ g ⨟ h" is "f ⨟ (g ⨟ h)".

In semigroupoids, we define the identity property as conjunction (implemented as pair type _×_) of the two one-sided identity properties; the pair components will later be extracted using the projections proj_1 and proj_2:

```
isLeftIdentity isRightIdentity isIdentity : {A : Obj}
                                            → Mor A A → Set (ℓi ⊔ ℓj ⊔ ℓk)
isLeftIdentity  {A} I = {B : Obj} {R : Mor A B} → I ⨟ R ≈ R
isRightIdentity {A} I = {B : Obj} {R : Mor B A} → R ⨟ I ≈ R
isIdentity        I = isLeftIdentity I × isRightIdentity I
```

[1] We can write f ≈ g for two morphisms from carrier set Mor A B of the hom-setoid Hom A B since the object arguments A and B of _≈_ are declared implicit and can be derived from the type of f and g.

The type of congruence of composition, ⨟-cong, is declared using a *telescope* introducing the seven named arguments A, B, C, f_1, f_2, g_1, and g_2 (here all implicit), which can be referred to in later parts of the type. The resulting *dependent function type* corresponds to "dependent products" frequently written using Π in other presentations of type theory.

A Category has the following additional **field**s for identity morphisms and identity properties:

```
field Id      : {A : Obj} → Mor A A
      leftId  : {A : Obj} → isLeftIdentity  (Id {A})
      rightId : {A : Obj} → isRightIdentity (Id {A})
```

As context for the remaining sections, we now show a monolithic definition of ordered semigroupoids with converse (OSGC). As argued in [Kah04], we approach allegories and Kleene categories via common primitives providing a local ordering on homsets. Restricting ourselves to this common core turns out to be sufficient for the current paper, where we will *not* need to add meets or joins in the local homsets orderings.

In locally ordered categories, "homsets" are partial orders, so an OSGC first of all contains a semigroupoid that uses for its "homsets" the underlying setoids. The local poset ordering relations are again collected into a global parameterised relation, $\sqsubseteq$. We also add the involutory converse operator $\_^\smile$ as a postfix operator, and give it higher precedence than all binary operators.

```
record OSGC′ {ℓi ℓj ℓk₁ ℓk₂ : Level} {Obj : Set ℓi}
             (Hom : Obj → Obj → Poset ℓj ℓk₁ ℓk₂)
             : Set (ℓi ⊔ ℓsuc (ℓj ⊔ ℓk₁ ⊔ ℓk₂)) where
  field semigroupoid : Semigroupoid′ (λ A B → posetSetoid (Hom A B))
  open Semigroupoid′ semigroupoid hiding (semigroupoid)
  infix 4 _⊑_; infix 10 _˘
  _⊑_ = λ {A} {B} → Poset._≤_ (Hom A B)
  field
    ⨟-monotone : {A B C : Obj} {f f' : Mor A B} {g g' : Mor B C}
                 → f ⊑ f' → g ⊑ g' → f ⨟ g ⊑ f' ⨟ g'
    _˘         : {A B : Obj}                    → Mor A B → Mor B A
    ˘˘         : {A B : Obj}   {R : Mor A B} → (R ˘) ˘ ≈ R
    ˘-involution : {A B C : Obj} {R : Mor A B} {S : Mor B C}
                                               → (R ⨟ S) ˘ ≈ S ˘ ⨟ R ˘
    ˘-monotone : {A B : Obj} {R S : Mor A B} → R ⊑ S → R ˘ ⊑ S ˘
```

Without identities, we frequently need the (one- and two-sided) sub- and superidentity properties, all of type $\{A : Obj\} \to (p : Mor\ A\ A) \to Set\ (\ell i \sqcup \ell j \sqcup \ell k_2)$:

```
isLeftSubidentity    {A} p = {B : Obj} {R : Mor A B} → p ⨟ R ⊑ R
isRightSubidentity   {A} p = {B : Obj} {S : Mor B A} → S ⨟ p ⊑ S
isSubidentity            p = isLeftSubidentity p × isRightSubidentity p
isLeftSuperidentity  {A} p = {B : Obj} {R : Mor A B} → R ⊑ p ⨟ R
isRightSuperidentity {A} p = {B : Obj} {S : Mor B A} → S ⊑ S ⨟ p
isSuperidentity          p = isLeftSuperidentity p × isRightSuperidentity p
```

With these, we can define, already in OSGCs, the following standard relation-algebraic properties, all of type $\{A\ B : Obj\} \to Mor\ A\ B \to Set\ (\ell i \sqcup \ell j \sqcup \ell k_2)$:

```
isUnivalent R = isSubidentity   (R ˘ ⨟ R)
isTotal     R = isSuperidentity (R ⨟ R ˘)
isMapping   R = isUnivalent R × isTotal R
isInjective R = isSubidentity   (R ⨟ R ˘)
isSurjective R = isSuperidentity (R ˘ ⨟ R)
isBijective R = isInjective R × isSurjective R
```

Total and univalent morphisms (in *Rel*, these are the total functions) are called *mappings*; for morphisms that are known to be mappings we define the dependent sum type Mapping containing the morphism and a proof of its mapping properties:

```
record Mapping (A B : Obj) : Set (ℓi ⊔ ℓj ⊔ ℓk₂) where
  field mor : Mor A B
        prf : isMapping mor
```

The mappings of an OSGC $\mathcal{S}$ form a semigroupoid MapSG $\mathcal{S}$ where the morphisms from A to B are the Mappings of $\mathcal{S}$, that is, Mor (MapSG $\mathcal{S}$) A B = Mapping $\mathcal{S}$ A B.

Adding the identities of Category to an OSGC results in an *ordered category with converse* (OCC); mappings of an OCC $\mathcal{C}$ form the category MapCat $\mathcal{C}$, and the OSGC versions of univalence, totality, ... are equivalent to the more habitual OCC versions R ˘ ⨟ R ⊑ Id etc.

In the remainder of this paper, and in the context of a given OSGS $\mathcal{S}$ or OCC $\mathcal{C}$, we append subscript "$_1$" to material taken from MapSG $\mathcal{S}$, respectively MapCat $\mathcal{C}$, in particular for equality _≈₁_ and composition _⨟₁_ of mappings.

4 Power Operators in Ordered Semigroupoids with Converse

In the following, the minimal setting is a ordered semigroupoid with converse (OSGC), with equality _≈_ and inclusion _⊑_ and composition _⨟_ of morphisms. In an OSGC, morphisms are naturally considered as a generalisation of *relations*, not just functions.

Total functions, called *mappings*, are a derived concept in OSGCs (see the end of the previous section); the induced semigroupoid of mappings (base morphisms together with univalence and totality proofs) has equality _≈₁_ and composition _⨟₁_.

A *power operator* consists of the following items:

```
ℙ : Obj → Obj                                    -- power object operator
∈ : {A : Obj} → Mor A (ℙ A)                      -- membership "relation"
Λ : {A B : Obj} → Mor A B → Mapping A (ℙ B)      -- "power transpose"
```

"Power transpose" maps a "relation" R : Mor A B to a "set-valued function" Mapping A (ℙ B).

The following axioms need to be satisfied; these are the two sides of one logical equivalence used by Bird and de Moor [BdM97, Sect. 4.6] to axiomatize the power allegories of Freyd and Scedrov [FS90, 2.4]:

$$\begin{aligned}
&\Lambda{\Rightarrow}\in : \{A\ B : \mathsf{Obj}\}\ \{R : \mathsf{Mor}\ A\ B\}\ \{f : \mathsf{Mapping}\ A\ \mathbb{P}B\} \\
&\qquad \to f \approx_1 \Lambda\ R \\
&\qquad \to \mathsf{Mapping.mor}\ f \fatsemi \in^{\smile} \approx R \\
&\in{\Rightarrow}\Lambda : \{A\ B : \mathsf{Obj}\}\ \{R : \mathsf{Mor}\ A\ B\}\ \{f : \mathsf{Mapping}\ A\ \mathbb{P}B\} \\
&\qquad \to \mathsf{Mapping.mor}\ f \fatsemi \in^{\smile} \approx R \\
&\qquad \to f \approx_1 \Lambda\ R
\end{aligned}$$

Throughout this paper, we will use the convention that subscript "$_0$" abbreviates an application of Mapping.mor, but we explicitly show only this first definition following this pattern:

$$\begin{aligned}
&\Lambda_0 : \mathsf{Mor}\ A\ B \to \mathsf{Mor}\ A\ \mathbb{P}B \\
&\Lambda_0\ R\ =\ \mathsf{Mapping.mor}\ (\Lambda\ R)
\end{aligned}$$

From the power axioms, we derive the following laws (given objects A and B); the first is used as axiom by Freyd and Scedrov [FS90, 2.4]:

$$\begin{aligned}
&\Lambda\fatsemi\in^{\smile} &&: \{R : \mathsf{Mor}\ A\ B\} && \to \Lambda_0\ R \fatsemi \in^{\smile} \approx R \\
&\Lambda\text{-}\fatsemi\in^{\smile} &&: \{f : \mathsf{Mapping}\ A\ (\mathbb{P}\ B)\} && \to \Lambda\ (\mathsf{Mapping.mor}\ f \fatsemi \in^{\smile}) \approx_1 f \\
&\Lambda\text{-cong} &&: \{R_1\ R_2 : \mathsf{Mor}\ A\ B\} && \to R_1 \approx R_2 \to \Lambda\ R_1 \approx_1 \Lambda\ R_2 \\
&{\smile}\fatsemi\Lambda &&: \{R : \mathsf{Mor}\ A\ B\} && \to R^{\smile} \fatsemi \Lambda_0\ R \sqsubseteq \in
\end{aligned}$$

We can define the function that returns, for each "set", the set containing all its "elements":

$$\begin{aligned}
&\mathsf{Id}\mathbb{P} : \{A : \mathsf{Obj}\} \to \mathsf{Mapping}\ (\mathbb{P}\ A)\ (\mathbb{P}\ A) \\
&\mathsf{Id}\mathbb{P}\ =\ \Lambda\ (\in^{\smile})
\end{aligned}$$

If there is an identity "relation" on $\mathbb{P}\ A$, then $\mathsf{Id}\mathbb{P}\ \{A\}$ is equal to that identity, as one would expect. However, without assuming identities, we only succeeded to show that $\mathsf{Id}\mathbb{P}\ \{A\}$ is a right-identity for mappings.

For any two power operators, we obtain mappings between $\mathbb{P}_1\ A$ and $\mathbb{P}_2\ A$ that compose to $\mathsf{Id}\mathbb{P}_1$, respectively $\mathsf{Id}\mathbb{P}_2$. Therefore, if the base OSGC has identities, then that makes the two power operators isomorphic, and more generally, we have that power operators in OCCs are unique up to natural isomorphisms.

In the context of a power operator, a "power order" is an indexed relation on power objects satisfying conditions appropriate for a "subset relation":

$$\begin{aligned}
&\textbf{record}\ \mathsf{IsPowerOrder}\ (\Omega : \{A : \mathsf{Obj}\} \to \mathsf{Mor}\ (\mathbb{P}\ A)\ (\mathbb{P}\ A)) \\
&\qquad\qquad : \mathsf{Set}\ (\ell i \sqcup \ell j \sqcup \ell k_1 \sqcup \ell k_2)\ \textbf{where} \\
&\quad \textbf{field}\ \in\fatsemi\Omega &&: \{A : \mathsf{Obj}\} && \to \in \fatsemi\ \Omega\ \{A\} \sqsubseteq \in \\
&\qquad \Omega\text{-universal} &&: \{A : \mathsf{Obj}\}\ \{R : \mathsf{Mor}\ (\mathbb{P}\ A)\ (\mathbb{P}\ A)\} && \to \in \fatsemi\ R \sqsubseteq \in \to R \sqsubseteq \Omega
\end{aligned}$$

(The first condition $\in \fatsemi\ \Omega \sqsubseteq \in$ could be replaced with the converse implication of the second, namely $R \sqsubseteq \Omega \to \in \fatsemi\ R \sqsubseteq \in$.) A power operator together with a power order gives rise to existence of all right residuals (which, with converse, in turn implies existence of all left residuals), see Appendix A for the proof.

5 Power Orders via Residuals

The *right-residual* Q \ S, "Q under S", of two morphisms Q : Mor A B and S : Mor A C is the largest solution in X of the inclusion Q ⨾ X ⊑ S; formally, it is defined by:

```
_\_ : {A B C : Obj} → Mor A B → Mor A C → Mor B C
\-cancel-outer : {A B C : Obj} {S : Mor A C} {Q : Mor A B} → Q ⨾ (Q \ S) ⊑ S
\-universal :    {A B C : Obj} {S : Mor A C} {Q : Mor A B} {R : Mor B C}
              → Q ⨾ R ⊑ S → R ⊑ Q \ S
```

The last of these can be understood as an implication axiom \-universal, stating: "If Q ⨾ R ⊑ S, then R ⊑ Q \ S". Technically, it is a function taking, after six implicit arguments, one explicit argument, say "p", of type Q ⨾ R ⊑ S, and then the application \-universal p is of type (i.e., a proof for) R ⊑ Q \ S. For concrete relations, Q \ S relates b with c if and only if for all a with aQb we have aSc.

In the presence of converse, right residuals produce left residuals (S / R, "S over R") and vice versa, so it does not matter whether we assume one or both.

Adding residuals to the base OSGC enables the standard definition of the "set" inclusion relation [BSZ89, FS90] (for which we also easily show IsPowerOrder Ω):

```
Ω : {A : Obj} → Mor (ℙ A) (ℙ A)
Ω = ∈ \ ∈
```

This is transitive, and "as reflexive as we can state" without identities:

```
Ω-trans : {A : Obj} → Ω ⨾ Ω ⊑ Ω {A}
Ω-trans = \-cancel-middle

Id℘⊑Ω : {A : Obj} → Mapping.mor Id℘ ⊑ Ω {A}
Id℘⊑Ω′ = \-universal (⊑-begin
    ∈ ⨾ Id℘₀
  ≈˘⟨ ⨾-cong₁ ˘˘ ⟩
    (∈ ˘) ˘ ⨾ Λ₀ (∈ ˘)
  ⊑⟨ ˘⨾Λ ⟩
    ∈
  □)
```

(Here the argument proof to \-universal is presented in *calculational style*, which technically uses mixfix operators ⊑-begin_, _≈˘⟨_⟩_, _⊑⟨-⟩_, _□ with carefully arranged precedences to produce a fully formal "proof term with type annotations". This technique goes back to Augustsson and Norell [Aug99, Nor07], and in its present form essentially comes from Danielsson's Agda standard library [D+13]. The first step above contains an application of ˘˘ : (R ˘) ˘ ≈ R, applied via ⨾-cong₁ at the first argument of the composition, but in *backwards* direction, which is expressed by the ˘ in _≈˘⟨_⟩_. The expansion of the definition of Id℘₀, that also happens in the first step, does not need to be mentioned, since for Agda, both expressions are the same via normalisation.)

The following property, shown using residual and power properties, will be useful below:

```
Λ₀⨟Ω˘ : {A B : Obj} {R : Mor A B} → Λ₀ R ⨟ Ω ˘ ≈ R / ∈ ˘
Λ₀⨟Ω˘ {R = R} = ≈-begin
    Λ₀ R ⨟ (∈ \ ∈) ˘
  ≈⟨ ⨟-cong₂ \-˘ ⟩
    Λ₀ R ⨟ (∈ ˘ / ∈ ˘)
  ≈⟨ /-outer-⨟-≈ Λ-mapping ⟩
    (Λ₀ R ⨟ ∈ ˘) / ∈ ˘
  ≈⟨ /-cong₁ Λ⨟∈˘ ⟩
    R / ∈ ˘                                  □
```

6 Contexts in OSGCs with Powers and Residuals

A context in our abstract setting consists of two objects together with a morphism of the base "relation"-OSGC:

```
record AContext : Set (ℓi ⊔ ℓj) where
  field ent : Obj            -- "entities"
        att : Obj            -- "attributes"
        inc : Mor ent att    -- "incidence"
```

In such a context, the incidence "relation" inc induces "concepts" as sets $\mathsf{inc} \uparrow p$ of attributes shared by a set $p : \mathbb{P}\ \mathsf{ent}$ of entities, and sets $\mathsf{inc} \downarrow q$ of entities sharing all attributes in $q : \mathbb{P}\ \mathsf{att}$, set-theoretically defined in the following way:

$$\mathsf{inc} \uparrow p = \{a : \mathsf{att} \mid \forall e \in p \;.\; e\ \mathsf{inc}\ a\} \qquad \text{and} \qquad \mathsf{inc} \downarrow q = \{e : \mathsf{ent} \mid \forall a \in q \;.\; e\ \mathsf{inc}\ a\} \;.$$

We define the general operators _↑ and _↓ as postfix operators, so they need to be separated from their argument by a space:

```
_↑ : {A B : Obj} → Mor A B → Mapping (ℙ A) (ℙ B)
R ↑ = Λ (∈ \ R)
_↓ : {A B : Obj} → Mor A B → Mapping (ℙ B) (ℙ A)
R ↓ = Λ (∈ \ (R ˘))
```

The fact that these form a Galois connection, set-theoretically

$$p \subseteq \mathsf{R} \downarrow q \qquad \Leftrightarrow \qquad q \subseteq \mathsf{R} \uparrow p \qquad\qquad \text{for all } p : \mathbb{P}\ \mathsf{A} \text{ and } q : \mathbb{P}\ \mathsf{B},$$

can now be stated as a simple morphism equality and shown by algebraic calculation using residual and power properties:

```
Galois-↓-↑ : {A B : Obj} {R : Mor A B} → Ω ⨟ (R ↓₀) ˘ ≈ R ↑₀ ⨟ Ω ˘
Galois-↓-↑ {A} {B} {R} = ≈-begin
    Ω ⨟ Λ₀ (∈ \ R ˘) ˘
```

```
≈˘⟨ ˘-involutionRightConv ⟩
  (Λ₀ (∈ \ R ˘) ⨟ Ω ˘) ˘
≈⟨ ˘-cong Λ₀⨟Ω˘ ⟩
  ((∈ \ R ˘) / ∈ ˘) ˘
≈⟨ /˘-˘ ⟩
  ∈ \ (∈ \ R ˘) ˘
≈⟨ \-cong₂ \˘-˘ ⟩
  ∈ \ (R / ∈ ˘)
≈⟨ \/-≈ ⟩
  (∈ \ R) / ∈ ˘
≈˘⟨ Λ₀⨟Ω˘ ⟩
  Λ₀ (∈ \ R) ⨟ Ω ˘                □
```

For the composed operators _↑↓ and _↓↑, with R ↑↓ = R ↑ ⨟₁ R ↓ and R ↓↑ = R ↓ ⨟₁ R ↑, the closure properties follow via further, partially lengthy calculations.

For a "set-valued *relation*" R : Mor X (ℙ A), the "set-valued function" Lub R can be understood as mapping each "element" x of X to the union of all sets that R relates with x; this union contains an "element" a of A if and only if R ⨟ ∈ ˘ relates x with a. From this "relation", Lub R is obtained as its power transpose, and similarly Glb for the intersection instead of union:

```
Lub Glb : {X A : Obj} (R : Mor X (ℙ A)) → Mapping X (ℙ A)
Lub R = Λ (R ⨟ ∈ ˘)
Glb R = Λ (R ˘ \ ∈ ˘)
```

The properties of "mapping unions to intersections" and vice versa can now be defined as follows, for objects A and B, and f : Mapping (ℙ B) (ℙ A):

```
Lub-cocontinuous f = ∀ {X : Obj} (Q : Mor X (ℙ B))
                     → Lub Q ⨟₁ f ≈₁ Glb (Q ⨟ Mapping.mor f)
Glb-cocontinuous f = ∀ {X : Obj} (Q : Mor X (ℙ B))
                     → Glb Q ⨟₁ f ≈₁ Lub (Q ⨟ Mapping.mor f)
```

Both of the operators _↑ and _↓ are Lub-cocontinuous, as can be shown in somewhat lengthy calculations, of which we show only the first — this contains three levels of nested calculations, indented precisely according to level, and uses in the reasons several different infix transitivity combinators following a common naming pattern, including _⟨≈˘⊑⟩_ : {...} → y ≈ x → y ⊑ z → x ⊑ z:

```
↓-Lub-cocontinuous : {A B : Obj} (R : Mor A B) → Lub-cocontinuous (R ↓)
↓-Lub-cocontinuous R {X} Q = ≈-begin
    Λ₀ (Q ⨟ ∈ ˘) ⨟ Λ₀ (∈ \ (R ˘))
  ≈⟨ ∈⇒Λ {f = Λ (Q ⨟ ∈ ˘) ⨟₁ Λ (∈ \ (R ˘))} (≈-begin
       (Λ₀ (Q ⨟ ∈ ˘) ⨟ Λ₀ (∈ \ (R ˘))) ⨟ ∈ ˘
     ≈⟨ ⨟-assoc ⟨≈≈⟩ ⨟-cong₂ Λ⨟∈˘ ⟩
       Λ₀ (Q ⨟ ∈ ˘) ⨟ (∈ \ (R ˘))
     ≈⟨ \-inner-⨟ Λ-mapping ⟩
```

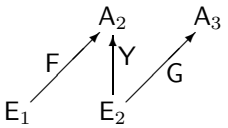
```
    (∈ ⨾ Λ₀ (Q ⨾ ∈ ˘) ˘) \ (R ˘)
≈⟨ \-cong₁ (˘-involutionRightConv ⟨≈˘≈⟩ ˘-cong Λ⨾∈˘) ⟨≈≈˘⟩ /-˘ ⟩
    (R / (Q ⨾ ∈ ˘)) ˘
≈⟨ ˘-cong (⊑-antisym
    (/-universal (⊑-begin
          (R / (Q ⨾ ∈ ˘)) ⨾ Q ⨾ Λ₀ (∈ \ R ˘)
      ⊑⟨ ⨾-assocL ⟨≈⊑⟩ ⨾-monotone₁ /-cancel-⨾-inner ⟩
          (R / ∈ ˘) ⨾ Λ₀ (∈ \ R ˘)
      ⊑⟨ ⨾-cong₁ \˘-˘ ⟨≈˘⊑⟩ ˘⨾Λ ⟩
          ∈
      □))
    (/-universal (⊑-begin
          (∈ / (Q ⨾ Λ₀ (∈ \ R ˘))) ⨾ (Q ⨾ ∈ ˘)
      ⊑⟨ ⨾-assocL ⟨≈⊑⟩ ⨾-monotone₁ /-cancel-⨾-inner ⟩
          (∈ / Λ₀ (∈ \ R ˘)) ⨾ ∈ ˘
      ⊑⟨ ⨾-monotone₂ (proj₁ Λ-total ⟨⊑≈⟩ ⨾-assoc) ⟩
          (∈ / Λ₀ (∈ \ R ˘)) ⨾ Λ₀ (∈ \ R ˘) ⨾ Λ₀ (∈ \ R ˘) ˘ ⨾ ∈ ˘
      ⊑⟨ ⨾-assocL ⟨≈⊑⟩ ⨾-monotone₁ /-cancel-outer ⟩
          ∈ ⨾ Λ₀ (∈ \ R ˘) ˘ ⨾ ∈ ˘
      ≈⟨ ⨾-assocL ⟨≈≈˘⟩ ⨾-cong₁ ˘-involutionRightConv ⟩
          (Λ₀ (∈ \ R ˘) ⨾ ∈ ˘) ˘ ⨾ ∈ ˘
      ≈⟨ ⨾-cong₁ (˘-cong Λ⨾∈˘) ⟩
          (∈ \ R ˘) ˘ ⨾ ∈ ˘
      ⊑⟨ ⨾-cong₁ \˘-˘ ⟨≈⊑⟩ /-cancel-outer ⟩
          R
      □))) ⟩
    (∈ / (Q ⨾ Λ₀ (∈ \ R ˘))) ˘
≈⟨ /-˘ ⟩
    (Q ⨾ Λ₀ (∈ \ (R ˘))) ˘ \ ∈ ˘
□) ⟩
  Λ₀ ((Q ⨾ Λ₀ (∈ \ (R ˘))) ˘ \ ∈ ˘)
□
```

However, the closest we can have to Glb-cocontinuous (R ↑) is the following (where the final → has also been proven in the opposite direction):

```
↑-Glb-cocontinuous : {A B X : Obj} (R : Mor A B) (Q : Mor X (ℙ A))
                   → (∈ / Q) \ R ≈ (Q ⨾ (∈ \ R))
                   → Glb Q ⨾₁ (R ↑) ≈₁ Lub (Q ⨾ Mapping.mor (R ↑))
```

The reason for the general failure of Glb-cocontinuity is that if Q is not total, the resulting empty intersections on the left-hand side may be mapped by R ↑ to arbitrary sets, but on the right-hand side, the resulting empty unions are always the empty set. In particular in the power-allegory induced by any topos, if we set Q = ⊥ (the "empty relation") and R = ⊤ (the "universal relation"), then we have: $(\in / Q) \setminus R \quad \approx \quad (\in / \bot) \setminus \top \quad \approx \quad \top \quad \not\approx \quad \bot \quad \approx \quad \bot \fatsemi (\in \setminus \top) \quad \approx \quad Q \fatsemi (\in \setminus R)$

For composition of context homomorphisms, we will require Lub-cocontinuity of $\mathsf{G}\downarrow \fatsemi_1 \mathsf{Y}\uparrow \fatsemi_1 \mathsf{F}\downarrow$ in the following situation:



Due to the fact that $\mathsf{Y}\uparrow$ is not necessarily Glb-cocontinuous, context homomorphisms require additional "compatibility" conditions to enable the following calculation, which closely follows [Mos13], to go through:

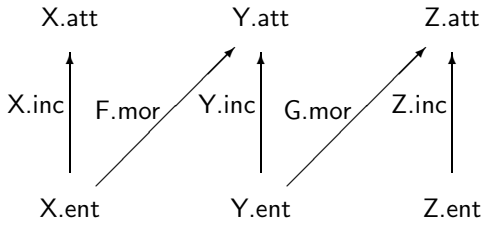
```
↓↑↓-Lub-cocontinuous : {E₁ E₂ A₂ A₃ : Obj}
                      → (F : Mor E₁ A₂) (Y : Mor E₂ A₂) (G : Mor E₂ A₃)
                      → (F-trgCompat : Y ↓↑ ⨟₁ F ↓ ≈₁ F ↓)
                      → (G-srcCompat : G ↓ ⨟₁ Y ↑↓ ≈₁ G ↓)
                      → Lub-cocontinuous (G ↓ ⨟₁ Y ↑ ⨟₁ F ↓)
↓↑↓-Lub-cocontinuous F Y G F-trgCompat G-srcCompat Q = ≈₁-begin
    Lub Q ⨟₁ G ↓ ⨟₁ Y ↑ ⨟₁ F ↓
  ≈₁⟨ ⨟-assocL ⟨≈≈⟩ ⨟-cong₁ (↓-Lub-cocontinuous G Q) ⟩
    Glb (Q ⨟ G ↓₀) ⨟₁ Y ↑ ⨟₁ F ↓
  ≈₁⟨ ⨟-cong₁ (Glb-cong (⨟-cong₂ G-srcCompat ⟨≈˘≈⟩ ⨟-assocL₃₊₁)) ⟩
    Glb ((Q ⨟ G ↓₀ ⨟ Y ↑₀) ⨟ Y ↓₀) ⨟₁ Y ↑ ⨟₁ F ↓
  ≈₁⟨ ⨟-cong₁ (↓-Lub-cocontinuous Y (Q ⨟ G ↓₀ ⨟ Y ↑₀)) ⟨≈˘≈⟩ ⨟-assoc ⟩
    Lub (Q ⨟ G ↓₀ ⨟ Y ↑₀) ⨟₁ Y ↓ ⨟₁ Y ↑ ⨟₁ F ↓
  ≈₁⟨ ⨟-cong₂ (⨟-assocL ⟨≈≈⟩ F-trgCompat) ⟩
    Lub (Q ⨟ G ↓₀ ⨟ Y ↑₀) ⨟₁ F ↓
  ≈₁⟨ ↓-Lub-cocontinuous F (Q ⨟ G ↓₀ ⨟ Y ↑₀) ⟨≈≈⟩ Glb-cong ⨟-assoc₃₊₁ ⟩
    Glb (Q ⨟ G ↓₀ ⨟ Y ↑₀ ⨟ F ↓₀)
  □₁
```

A context homomorphism, following Moshier [Mos13] and Jipsen [Jip12], includes the compatibility properties used above. In order to be able to refer to the **field**s of the source and target contexts X and Y with qualified names like X.ent instead of AContext.ent X, we need to use the "module nature" of **record**s in Agda and define local module names for X and Y. (In the following, we will omit these local module definitions for the sake of brevity, and since there will be no danger of confusion.)

```
record AContextHom (X Y : AContext) : Set (ℓi ⊔ ℓj ⊔ ℓk₁ ⊔ ℓk₂) where
  private module X = AContext X
          module Y = AContext Y
  field   mor : Mor X.ent Y.att
          srcCompat : mor ↓ ⨟₁ X.inc ↑↓ ≈₁ mor ↓
          trgCompat : Y.inc ↓↑ ⨟₁ mor ↓ ≈₁ mor ↓
```

If we now have three contexts X, Y, and Z connected by two context homomorphisms F : AContextHom X Y and G : AContextHom Y Z, then we define:

```
G↓⨟Y↑⨟F↓ : Mapping (ℙ Z.att) (ℙ X.ent)
G↓⨟Y↑⨟F↓ = G.mor ↓ ⨟₁ Y.inc ↑ ⨟₁ F.mor ↓
```



Applying ↓↑↓-Lub-cocontinuous, we obtain that the mapping G↓⨟Y↑⨟F↓ is Lub-cocontinuous just like F.mor ↓ : Mapping (ℙ Y.att) (ℙ X.ent) and G.mor ↓ : Mapping (ℙ Z.att) (ℙ Y.ent), but we are still missing a way to extract a _↓-preimage of type Mor X.ent Z.att.

7 Abstract Context Categories in OCCs with Powers and Residuals

It turns out that adding identities is sufficient for obtaining a partial inverse to the operator _↓. The key is that Λ Id : Mapping A (ℙ A) can be understood as mapping each "element" a : A to the singleton "set" $\{a\}$: ℙ A.

The "relation" singletons A relates a "subset of A" with all singletons contained in it:

```
singletons : {A : Obj} → Mor (ℙ A) (ℙ A)
singletons = ∈˘ ⨟ Λ₀ Id
```

Applying Lub to this produces the identity mapping on ℙ A:

```
Lub-singletons : {A : Obj} → Lub (singletons {A}) ≈₁ Id₁ {ℙ A}
Lub-singletons {A} = ≈₁-begin
    Λ ((∈˘ ⨟ Λ₀ Id) ⨟ ∈˘)
  ≈₁⟨ Λ-cong (⨟-assoc ⟨≈≈⟩ ⨟-cong₂ Λ⨟∈˘) ⟩
    Λ (∈˘ ⨟ Id {A})
  ≈₁⟨ Λ-cong (rightId ⟨≈≈˘⟩ leftId) ⟨≈≈⟩ Λ-⨟∈˘ {f = Id₁ {ℙ A}} ⟩
    Id₁ {ℙ A}                                                □₁
```

The operator [_] has the opposite type of _↓, and [f] relates a with b if and only if $a \in f\{b\}$:

```
[_] : {A B : Obj} → Mapping (ℙ B) (ℙ A) → Mor A B
[ f ] = (Λ₀ Id ⨟ Mapping.mor f ⨟ ∈˘) ˘
```

We always have [R↓] ≈ R:

```
[↓] : {A B : Obj} (R : Mor A B) → [ R↓ ] ≈ R
[↓] R = ≈-begin
    (Λ₀ Id ⨟ Λ₀ (∈ \ (R ˘)) ⨟ ∈ ) ˘
  ≈⟨ ˘-cong (⨟-cong₂ Λ⨟∈˘) ⟩
    (Λ₀ Id ⨟ (∈ \ (R ˘))) ˘
  ≈⟨ ˘-cong (\-inner-⨟ Λ-mapping) ⟩
    ((∈ ⨟ (Λ₀ Id) ˘) \ (R ˘)) ˘
  ≈⟨ \˘-˘ ⟩
    R / ((∈ ⨟ (Λ₀ Id) ˘) ˘)
  ≈⟨ /-cong₂ ˘-involutionRightConv ⟩
    R / (Λ₀ Id ⨟ ∈ ˘)
  ≈⟨ /-cong₂ Λ⨟∈˘ ⟨≈≈⟩ /-Id ⟩
    R
  □
```

For the opposite composition, [f]↓ ≈₁ f, we need Lub-cocontinuity of f:

```
[ ]↓ : {A B : Obj} (f : Mapping (ℙ B) (ℙ A))
    → Lub-cocontinuous f → [ f ] ↓ ≈₁ f
[ ]↓ f f-cocontinuous = ≈₁-begin
    [ f ] ↓
  ≈₁⟨ ≈-refl ⟩
    Λ (∈ \ ([ f ] ˘))
  ≈₁⟨ Λ-cong (\-cong₂ (˘˘ ⟨≈≈⟩ ⨟-assocL)) ⟩
    Λ (∈ \ (Mapping.mor (Λ Id ⨟₁ f) ⨟ ∈ ˘))
  ≈₁˘⟨ Λ-cong (\-cong₂ (⨟-cong₁ ˘˘)) ⟩
    Λ (∈ \ ((Λ₀ Id ⨟ Mapping.mor f) ˘ ˘ ⨟ ∈ ˘))
  ≈₁˘⟨ Λ-cong (\-cong₁ ˘-involutionLeftConv
                ⟨≈≈⟩ \-flip (˘-isBijective (Mapping.prf (Λ Id ⨟₁ f)))) ⟩
    Λ ((∈ ˘ ⨟ Λ₀ Id ⨟ Mapping.mor f) ˘ \ ∈ ˘)
  ≈₁˘⟨ Λ-cong (\-cong₁ (˘-cong ⨟-assoc)) ⟩
    Glb (singletons ⨟ Mapping.mor f)
  ≈₁˘⟨ f-cocontinuous singletons ⟩
    Lub singletons ⨟₁ f
  ≈₁⟨ ⨟-cong₁ Lub-singletons ⟨≈≈⟩ leftId ⟩
    f
  □₁
```

The last two steps represent the argument of [Mos13] that "If f sends unions to intersections, its behavior is determined by its behavior on singletons."

Using the instance

```
[ ⨟ ]↓ : [ G↓⨟Y↑⨟F↓ ] ↓ ≈₁ G↓⨟Y↑⨟F↓
[ ⨟ ]↓ = [ ]↓ G↓⨟Y↑⨟F↓
  (↓↑↓-Lub-cocontinuous F.mor Y.inc G.mor F.trgCompat G.srcCompat)
```

for the morphism composition at the end of Sect. 6, we obtain well-definedness:

```
_⨟⨟_ : (F : AContextHom X Y) (G : AContextHom Y Z) → AContextHom X Z
F ⨟⨟ G = record
  {mor = [ G↓⨟Y↑⨟F↓ ]
  ;srcCompat = ≈₁-begin
       [ G↓⨟Y↑⨟F↓ ] ↓ ⨟₁ X.inc ↑ ⨟₁ X.inc ↓
    ≈₁⟨ ⨟-cong₁ ([ ⨟ ]↓ F G) ⟨≈≈⟩ ⨟-assoc₃₊₁ ⟩
       G.mor ↓ ⨟₁ Y.inc ↑ ⨟₁ F.mor ↓ ⨟₁ X.inc ↑ ⨟₁ X.inc ↓
    ≈₁⟨ ⨟-cong₂₂ F.srcCompat ⟩
       G.mor ↓ ⨟₁ Y.inc ↑ ⨟₁ F.mor ↓
    ≈₁˘⟨ [ ⨟ ]↓ F G ⟩
       [ G↓⨟Y↑⨟F↓ ] ↓
    □₁
  ;trgCompat = ...  -- analogously
  }
```

Context homomorphism equality F ≋ G is defined as the underlying morphism equality F.mor ≈ G.mor. The left- and right-identity properties of the composition _⨟⨟_ reduce, via [↓], to srcCompat respectively trgCompat due to the fact that the identity context homomorphism on X has X.inc as mor, and the associativity proof also turns into a surprisingly short calculation:

$$X_1 \xrightarrow{F} X_2 \xrightarrow{G} X_3 \xrightarrow{H} X_4$$

```
ACH-assoc : ... → (F ⨟⨟ G) ⨟⨟ H ≋ F ⨟⨟ (G ⨟⨟ H)
ACH-assoc {X₁} {X₂} {X₃} {X₄} {F} {G} {H} = [ ]-cong
  {f₁ = H.mor ↓ ⨟₁ X₃.inc ↑ ⨟₁ FG.mor ↓}
  {f₂ = GH.mor ↓ ⨟₁ X₂.inc ↑ ⨟₁ F.mor ↓}
  (≈₁-begin
     H.mor ↓ ⨟₁ X₃.inc ↑ ⨟₁ FG.mor ↓
  ≈₁⟨ ⨟-cong₂₂ ([ ⨟ ]↓ F G) ⟩
     H.mor ↓ ⨟₁ X₃.inc ↑ ⨟₁ G.mor ↓ ⨟₁ X₂.inc ↑ ⨟₁ F.mor ↓
  ≈₁⟨ ⨟-assocL₃₊₁ ⟨≈≈˘⟩ ⨟-cong₁ ([ ⨟ ]↓ G H) ⟩
     GH.mor ↓ ⨟₁ X₂.inc ↑ ⨟₁ F.mor ↓
  □₁)
  where FG = F ⨟⨟ G ;  GH = G ⨟⨟ H
```

With this, a Category of AContexts with AContextHoms as morphisms is easily defined and checked by Agda.

8 Conclusion

A "natural", more direct formalisation of contexts would allow arbitrary Sets (or possibly Setoids) of entities and attributes, exactly as in the mathematical definition:

```
record Context (ℓe ℓa ℓr : Level) : Set (ℓsuc (ℓe ⊔ ℓa ⊔ ℓr)) where
  field ent : Set ℓe            -- "entities"
        att : Set ℓa            -- "attributes"
        inc : 𝓡el ℓr ent att    -- "incidence"
```

Although this has the advantage of additional universe polymorphism, it appears that the compatibility conditions force all the sets and relations to the same levels:

```
record Hom {ℓS ℓr : Level} (A B : Context ℓS ℓS ℓr) : Set (ℓS ⊔ ℓsuc ℓr) where
  private module A = Context A
          module B = Context B
  field   mor : 𝓡el ℓr A.ent B.att
          srcCompat : A.inc ↑↓ ∘∘ mor ↓ ≈⌊ ℙ B.att ℓr ⇨ ℙ A.ent _ ⌋ mor ↓
          trgCompat : mor ↓ ∘∘ B.inc ↓↑ ≈⌊ ℙ B.att ℓr ⇨ ℙ A.ent _ ⌋ mor ↓
```

An important disadvantage of this approach is that, for example, quotient contexts will have entity and attribute sets that lack many of the interfaces that would be useful for programming, for example serialisation.

In contrast, using our abstract approach makes it possible to instantiate the base OCC differently for different purposes:

- For theoretical investigations, using OCCs of setoids (or even of Sets) as defined in Relation.Binary.Heterogeneous.Categoric.OCC of the RATH-Agda libraries [Kah11b] provides all the flexibility of the general mathematical setting, but without useful execution mechanisms.
- For data processing applications, that is, "for programming", using implementation-oriented OCCs as for example that of SULists mentioned in [Kah12] provides additional interfaces and correct-by-construction executable implementations.

Beyond the theoretically interesting fact that context categories can be formalised in OCCs with residuals and powers, this paper also demonstrated that such an essentially theoretical development can be fully mechanised and still be presented in readable calculational style, where writing is not significantly more effort than a conventional calculational presentation in LaTeX.

In comparison with similar developments in Isabelle/HOL [Kah03], the use of Agda enables a completely natural mathematical treatment of categories, nested calculational proofs, and direct use of theories as modules of executable programs.

Acknowledgements. I am grateful to the anonymous referees for their constructive comments, and to Musa Al-hassy for numerous useful suggestions for improving readability.

References

[Aug99] Augustsson, L.: Equality proofs in Cayenne (1999), `http://tinyurl.com/Aug99eqproof` (accessed January 3, 2014)

[BSZ86] Berghammer, R., Schmidt, G., Zierer, H.: Symmetric Quotients. Technical Report TUM-INFO 8620, Technische Universität München, Fakultät für Informatik, 18 p. (1986)

[BSZ89] Berghammer, R., Schmidt, G., Zierer, H.: Symmetric Quotients and Domain Constructions. Inform. Process. Lett. 33, 163–168 (1989)

[BdM97] Bird, R.S., de Moor, O.: Algebra of Programming. International Series in Computer Science, vol. 100. Prentice Hall (1997)

[D+13] Danielsson, N.A. et al.: Agda Standard Library, Version 0.7 (2013), `http://tinyurl.com/AgdaStdlib`

[Ern05] Erné, M.: Categories of Contexts (2005) (preprint), `http://www.iazd.uni-hannover.de/~erne/preprints/CatConts.pdf`

[FS90] Freyd, P.J., Scedrov, A.: Categories, Allegories. North-Holland Mathematical Library, vol. 39. North-Holland, Amsterdam (1990)

[HKZ06] Hitzler, P., Krötzsch, M., Zhang, G.-Q.: A Categorical View on Algebraic Lattices in Formal Concept Analysis. Fund. Inform. 74, 301–328 (2006)

[Jip12] Jipsen, P.: Categories of Algebraic Contexts Equivalent to Idempotent Semirings and Domain Semirings. In: [KG12], pp. 195–206

[Kah03] Kahl, W.: Calculational Relation-Algebraic Proofs in Isabelle/Isar. In: Berghammer, R., Möller, B., Struth, G. (eds.) RelMiCS/AKA 2003. LNCS, vol. 3051, pp. 178–190. Springer, Heidelberg (2004)

[Kah04] Kahl, W.: Refactoring Heterogeneous Relation Algebras around Ordered Categories and Converse. J. Relational Methods in Comp. Sci. 1, 277–313 (2004)

[Kah08] Kahl, W.: Relational Semigroupoids: Abstract Relation-Algebraic Interfaces for Finite Relations between Infinite Types. J. Logic and Algebraic Programming 76, 60–89 (2008)

[Kah11a] Kahl, W.: Collagories: Relation-Algebraic Reasoning for Gluing Constructions. J. Logic and Algebraic Programming 80, 297–338 (2011)

[Kah11b] Kahl, W.: Dependently-Typed Formalisation of Relation-Algebraic Abstractions. In: de Swart, H. (ed.) RAMiCS 2011. LNCS, vol. 6663, pp. 230–247. Springer, Heidelberg (2011)

[Kah12] Kahl, W.: Towards Certifiable Implementation of Graph Transformation via Relation Categories. In: [KG12], pp. 82–97

[KG12] Kahl, W., Griffin, T.G. (eds.): RAMiCS 2012. LNCS, vol. 7560. Springer, Heidelberg (2012)

[Kah14] Kahl, W.: Relation-Algebraic Theories in Agda — RATH-Agda-2.0.0. Mechanically checked Agda theories available for download, with 456 pages literate document output (2014), `http://RelMiCS.McMaster.ca/RATH-Agda/`

[Mos13] Moshier, M.A.: A Relational Category of Polarities (2013) (unpublished draft)

[Nor07] Norell, U.: Towards a Practical Programming Language Based on Dependent Type Theory. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology (2007)

[Wil05] Wille, R.: Formal Concept Analysis as Mathematical Theory of Concepts and Concept Hierarchies. In: Ganter, B., Stumme, G., Wille, R. (eds.) Formal Concept Analysis. LNCS (LNAI), vol. 3626, pp. 1–33. Springer, Heidelberg (2005)

A Power Orders Give Rise to Right Residuals

In the context of an OSGC with a power operator, presence of a power implies that all right residuals exist, with $Q \setminus S = \Lambda_0\ (Q\ ^\smile) \fatsemi \Omega \fatsemi (\Lambda_0\ (S\ ^\smile))\ ^\smile$. We use the following additional power operator lemmas:

$\in\fatsemi\Lambda^\smile : \{R : \mathsf{Mor}\ A\ B\} \to \in \fatsemi (\Lambda_0\ R)\ ^\smile \approx R\ ^\smile$
$\fatsemi\Lambda\text{-}^\smile : \{Q : \mathsf{Mor}\ B\ A\} \to Q \fatsemi \Lambda_0\ (Q\ ^\smile) \sqsubseteq \in$

```
module PowerRightRes (Ω : {A : Obj} → Mor (ℙ A) (ℙ A))
 (isPowerOrder : IsPowerOrder Ω) where
 open IsPowerOrder isPowerOrder
 rightResOp : RightResOp orderedSemigroupoid
 rightResOp = record
   {_\_ = λ {A} {B} {C} Q S → Λ₀ (Q ˘) ⨟ Ω ⨟ (Λ₀ (S ˘)) ˘
   ; \-cancel-outer = λ {A} {B} {C} {S} {Q} → ⊑-begin
         Q ⨟ Λ₀ (Q ˘) ⨟ Ω ⨟ (Λ₀ (S ˘)) ˘
     ⊑⟨ ⨟-assocL ⟨≈⊑⟩ ⨟-monotone₁ ⨟Λ-˘ ⟩
         ∈ ⨟ Ω ⨟ (Λ₀ (S ˘)) ˘
     ⊑⟨ ⨟-assocL ⟨≈⊑⟩ ⨟-monotone₁ ∈⨟Ω ⟩
         ∈ ⨟ (Λ₀ (S ˘)) ˘
     ≈⟨ ∈⨟Λ˘ ⟨≈≈⟩ ˘˘ ⟩
         S
     □
   ; \-universal = λ {A} {B} {C} {S} {Q} {R} Q⨟R⊑S → ⊑-begin
         R
     ⊑⟨ proj₁ Λ-total ⟨⊑≈⟩ ⨟-assoc ⟩
         Λ₀ (Q ˘) ⨟ (Λ₀ (Q ˘)) ˘ ⨟ R
     ⊑⟨ ⨟-monotone₂₂ (proj₂ Λ-total) ⟩
         Λ₀ (Q ˘) ⨟ (Λ₀ (Q ˘)) ˘ ⨟ R ⨟ Λ₀ (S ˘) ⨟ (Λ₀ (S ˘)) ˘
     ⊑⟨ ⨟-monotone₂ (⨟-assocL₃₊₁ ⟨≈⊑⟩ ⨟-monotone₁ (Ω-universal (⊑-begin
           ∈ ⨟ (Λ₀ (Q ˘)) ˘ ⨟ R ⨟ Λ₀ (S ˘)
       ≈⟨ ⨟-assocL ⟨≈≈⟩ ⨟-cong₁ (∈⨟Λ˘ ⟨≈≈⟩ ˘˘) ⟩
```

```
                Q ⨟ R ⨟ Λ₀ (S ˘)
              ⊑⟨ ⨟-assocL ⟨≈⊑⟩ ⨟-monotone₁ Q⨟R⊑S ⟨⊑⊑⟩ ⨟Λ-˘ ⟩
                ∈
              □))) ⟩
            Λ₀ (Q ˘) ⨟ Ω ⨟ (Λ₀ (S ˘)) ˘
          □
      }
```

The standard definition of the power order via this right residual also returns the given power order: $\in \setminus \in \approx \Omega$.