Dependently-Typed Formalisation of Relation-Algebraic Abstractions

Wolfram Kahl

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

Abstract. We present a formalisation in the dependently-typed programming language Agda2 of basic category and allegory theory, and of generalised algebras where function symbols are interpreted in a parameter category. We use this nestable algebra construction as the basis for nestable category and allegory constructions, ultimately aiming at a formalised foundation of the algebraic approach to graph transformation, which uses constructions in categories of graph structures considered as unary algebras.

The features of Agda permit strongly-typed programming with these nested algebras and with relational homomorphisms between them in a natural mathematical style and with remarkable ease, far beyond what can be achieved even in Haskell.

Keywords: Dependently typed programming, algebras as data, allegories of relational algebra morphisms, nested algebras.

1 Introduction

In the context of computation, algebras are frequently seen as models of data *types*, with computations implementing their primitive and derived operations. However, algebras also have uses as data *values*, with computations producing new algebras from old. Examples for this are not only the Abstract State Machines (originally "Evolving Algebras") of Gurevich [13,14], but also any graph data structures, which can be considered as (typically unary) algebras. The "algebraic approach" to graph transformation [6] in particular takes that point of view, and applies abstractions from category theory to define and reason about graph transformation systems.

In this paper, we explore a flexible formalisation of aspects of relational categories and universal algebra in the dependently-typed programming language (and proof checker) Agda2 [26], leading up to allegories of "relational homomorphisms" between algebras, technically also known as bisimulations.

We start with an introduction to essential features of Agda2 (in the following just referred to as Agda) and its current standard library, and then (Sect. 3) summarise our formalisation of relation algebraic operations for the standard concept of relations in Agda. In Sect. 4 we turn to fine-grained, universe-polymorphic formalisations of categories and allegories, and elaborate more on the topics of domain (Sect. 5) and restricted residuals (Sect. 6). Our generalised formalisation of algebras is summarised in Sect. 7. We discuss some related work in Sect. 8.

H. de Swart (Ed.): RAMICS 2011, LNCS 6663, pp. 230–247, 2011.

The Agda theories discussed in this paper are available on-line at the URL http://RelMiCS.McMaster.ca/~kahl/RATH/Agda/.

2 Introduction to Agda: Types, Sets, Equality

The Agda home page[1] states:

> **Agda is a dependently typed functional programming language.** It has inductive families, i.e., data types which depend on values, such as the type of vectors of a given length. It also has parametrised modules, mixfix operators, Unicode characters, and an interactive Emacs interface which can assist the programmer in writing the program.
>
> **Agda is a proof assistant.** It is an interactive system for writing and checking proofs. Agda is based on intuitionistic type theory, a foundational system for constructive mathematics developed by the Swedish logician Per Martin-Löf. It has many similarities with other proof assistants based on dependent types, such as Coq, Epigram, Matita and NuPRL.

Syntactically and "culturally", Agda is quite close to Haskell. However, since Agda is strongly normalising and has no $\perp$ values, the underlying semantics is quite different. Also, 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 corresponds to the Haskell kind *; it is the type of all "normal" datatypes. For example, the Agda standard library defines the type Bool as follows:

```
data Bool : Set where true : Bool
                      false : Bool
```

Since Set needs again a type, there is Set_1, with $\mathsf{Set} : \mathsf{Set}_1$, etc., resulting in a hierarchy of "universes". Since Version 2.2.8, Agda supports *universe polymorphism*, with universes Set i where i is an element of the following special-purpose variant of the natural numbers:

```
data Level : Set where zero : Level
                       suc : (i : Level) → Level
```

With this, the conventional usage turns into syntactic sugar, so that Set is now Set zero, and Set_1 = Set (suc zero). The standard library uses "$\sqcup$" for maximum on Level; in our development, we systematically rename this to "$\uplus$", so that we use "$\sqcup$" as join in the inclusion order of morphisms, as customary in abstract relation algebra [28,27].

With universe polymorphism enabled, we may quantify over Level-typed variables that occur as Level arguments of Set. Universe polymorphism is essential for being able to talk about both "small" and "large" categories or relation algebras, or, for another example, also for being able to treat diagrams of graphs and graph homomorphisms as graphs again. We therefore use universe polymorphism throughout this paper.

[1] `http://wiki.portal.chalmers.se/agda/`

For example, the standard library includes the following definition for the universe-polymorphic parameterised Maybe type:

```
data Maybe {a : Level} (A : Set a) : Set a where just : (x : A) → Maybe A
                                                 nothing : Maybe A
```

Maybe has two parameters, a and A, where dependent typing is used since the type of the second parameter depends on the first parameter. The use of {...} flags a as an *implicit parameter* that can be elided where its type is implied by the call site of Maybe. This happens in the occurrences of Maybe A in the types of the data constructors just and nothing: In Maybe A, the value of the first, implicit parameter of Maybe can only be a, the level of the set A.

The same applies to implicit function arguments, and in most cases, implicit arguments or parameters are determined by later arguments respectively parameters. Frequently, implicit arguments correspond quite precisely to the implicit context of mathematical statements, so that the reader may be advised to skip implicit arguments at first reading of a type, and return to them for clarification where necessary for understanding the types of the explicit parameters.

While the Hindley-Milner typing of Haskell and ML allows function definitions without declaration of the function type, and type signatures without declaration of the universally quantified type variables, in Agda, all types and variables need to be declared, but implicit parameters and the type checking machinery used to resolve them alleviate that burden significantly. For example, the original definition writes only Maybe {a} (A : Set a) : Set a, since the type of a will be inferred from a's use as argument to Set. In this paper, we will rarely use this possibility to elide types of named arguments, since we estimate that the clarity of explicit typing is worth the additional "optical noise" especially for readers who are less familiar with Agda or dependently-typed theories.

The "programming types" like Maybe can be freely mixed with "formula types", inspired by the Curry-Howard-correspondence of "formulae as types, proofs as programs". The formula types of true formulae contain their proofs, while the formula types of false formulae are empty.

The standard library type of propositional equality has (besides two implicit parameters) one explicit parameter and one explicit argument; the definition therefore gives rise to types like the type "$2 \equiv 1 + 1$", which can be shown to be inhabited using the definition of natural numbers and natural number addition $+$, and the type "$2 \equiv 3$", which is an empty type, since it has no proof[2].

```
data _≡_ {a : Level} {A : Set a} (x : A) : A → Set a where refl : x ≡ x
```

The definition introduces types x ≡ y for any x and y of type A, but only the types x ≡ x are inhabited, and they contain the single element refl {a} {A} {x}.

In Agda, as in other type theories without quotient types, sets with equality are typically modelled as *setoids*, that is, carrier types equipped with an

[2] In Agda, almost all lexemes are separated by spaces, since almost all symbol combinations form legal names. Underscores as part of names indicate positions of explicit arguments for mixfix operators.

equivalence. This closely corresponds to the non-primitive nature of the "equality" test $(\equiv) : \mathsf{Eq}\ \mathsf{a} \Rightarrow \mathsf{a} \to \mathsf{a} \to \mathsf{Bool}$ in Haskell.

The standard library defines the following type of homogeneous relations:

```
Rel : {a : Level} → Set a → (l : Level) → Set (a ⊔ suc l)
Rel A l = A → A → Set l
```

A proof that _≈_ is an equivalence relation is a record containing the proofs of reflexivity, symmetry, and transitivity:

```
record IsEquivalence {a l : Level} {A : Set a} (_≈_ : Rel A l) : Set (a ⊔ l) where
  field refl  : {x : A} → x ≈ x
        sym   : {x y : A} → x ≈ y → y ≈ x
        trans : {x y z : A} → x ≈ y → y ≈ z → x ≈ z
```

A setoid is a dependent record consisting of a Carrier set, a relation _≈_ on that carrier, and a proof that that relation is an equivalence relation:

```
record Setoid c l : Set (suc (c ⊔ l)) where
  field Carrier : Set c
        _≈_     : Rel Carrier l
        isEquivalence : IsEquivalence _≈_
  open IsEquivalence isEquivalence public
```

An Agda record is also a module that may contain other material besides its **field**s; the "**open**" clause makes the fields of the equivalence proof available as if they were fields of Setoid. This language feature enables incremental extension of smaller theories to larger theories at very low notational cost.

The Preorder type of the Agda standard library adds a second preorder relation to a Setoid, with reflexivity with respect to the setoid equality; for a Poset, that preorder also needs to be antisymmetric with respect to the setoid equality.

3 Generalised Heterogeneous Concrete Relations

Concrete relations from a set A to another set B are normally defined to be the subsets of the Cartesian product $\mathsf{A} \times \mathsf{B}$. Equivalently, they can be seen as characteristic functions of type $(\mathsf{A} \times \mathsf{B}) \to \mathsf{Bool}$, or, in the curried variant, of type $\mathsf{A} \to \mathsf{B} \to \mathsf{Bool}$ (function type construction associates to the right). In Agda, it is more natural to replace Bool with a Set universe, with the understanding that R a b is the type of proofs that the pair (a, b) is in R, that is, R a b is empty if (a, b) is not in R, and inhabited if (a, b) is in R.

Therefore, we will use $\mathsf{R} : \mathsf{A} \to \mathsf{B} \to \mathsf{Set}$ for "small concrete" relations. This relation type, $\mathsf{A} \to \mathsf{B} \to \mathsf{Set}$, can also serve to represent other structures, for example, $\mathsf{G} : \mathsf{N} \to \mathsf{N} \to \mathsf{Set}$ could represent a graph G with node type N, and $\mathsf{G}\ \mathsf{n}_1\ \mathsf{n}_2$ would be the set of all edges from n_1 to n_2. Utilities defined for relations with types like $\mathsf{A} \to \mathsf{B} \to \mathsf{Set}$ therefore can be applied in many different contexts.

Since categories can be seen as graphs with additional structure, we obviously need full universe polymorphism, and define:

```
ℛel : {i j : Level} → (k : Level) → Set i → Set j → Set (i ⊎ j ⊎ suc k)
ℛel k A B = A → B → Set k
```

The order of parameters is a matter of taste; the standard library, which until recently defined only types of homogeneous relations, now defines the same types, but with a different argument order in the parameters of their constant REL. However, we have, by the definitions, type equality $\mathcal{R}el\ \mathsf{k}\ \mathsf{A}\ \mathsf{B} = \mathsf{REL}\ \mathsf{A}\ \mathsf{B}\ \mathsf{k}$, so that our $\mathcal{R}el$-based library is fully interoperable with the standard library, which does not provide typical relation-algebraic operations and laws. (The AoPA library of Mu *et al.* [25] does provide relation-algebraic operations and laws, but supports heterogeneous binary relations only at the levels 0 and, to a lesser degree, 1).

From this definition of relation types, we completely follow the standard procedure to formalisation of concrete relations in dependent type theory; inclusion, and equality of relations as derived from inclusion are defined as follows:

```
_⊆_ : {i j k₁ k₂ : Level} {A : Set i} {B : Set j}
      → ℛel k₁ A B → ℛel k₂ A B → Set (i ⊎ j ⊎ k₁ ⊎ k₂)
P ⊆ Q = ∀ {x y} → P x y → Q x y
_≑_ : {i j k₁ k₂ : Level} → {A : Set i} → {B : Set j}
      → ℛel k₁ A B → ℛel k₂ A B → Set (i ⊎ j ⊎ k₁ ⊎ k₂)
R ≑ S = (R ⊆ S) × (S ⊆ R)   -- × encodes logical conjunction ∧
```

Due to universe polymorphism, we could also have declared (equivalently):

```
_⊆_ : {i j k₁ k₂ : Level} {A : Set i} {B : Set j}
      → ℛel (i ⊎ j ⊎ k₁ ⊎ k₂) (ℛel k₁ A B) (ℛel k₂ A B)
```

These relations on $\mathcal{R}el$-relations are then used to define Setoids and Posets of $\mathcal{R}el$-relations. However, since $\mathcal{R}el$ deals directly with Sets, not with setoids, the identity relation has to be based on propositional equality, but that provided by the standard library, _≡_ presented in Sect. 2, forces the Level of its arguments onto its result, so we provide our own fully universe-polymorphic variant:

```
data _≡≡_ {k a} {A : Set a} (x : A) : A → Set k where ≡≡-refl : x ≡≡ x
```

With this, we can define universe-polymorphic identity relations:

```
idR : {k i : Level} {A : Set i} → ℛel k A A
idR = _≡≡_
```

In the module hierarchy Relation.Binary.Heterogeneous, we define standard relation-algebraic operations and properties, and prove the relevant laws in particular to be able to implement the abstract theories presented in Sect. 4.

4 Semigroupoids, Categories, Allegories, Collagories

We present a relatively fine-grained modularisation of sub-theories of distributive 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 [19], and on collagories as foundation for relation-algebraic graph transformation [20].

Semigroupoids are to categories as semigroups are to monoids — no identities are assumed. The following definition is taken from [19], and will probably appear to be quite conventional to readers familiar with the basics of category theory (except perhaps for the argument order of composition):

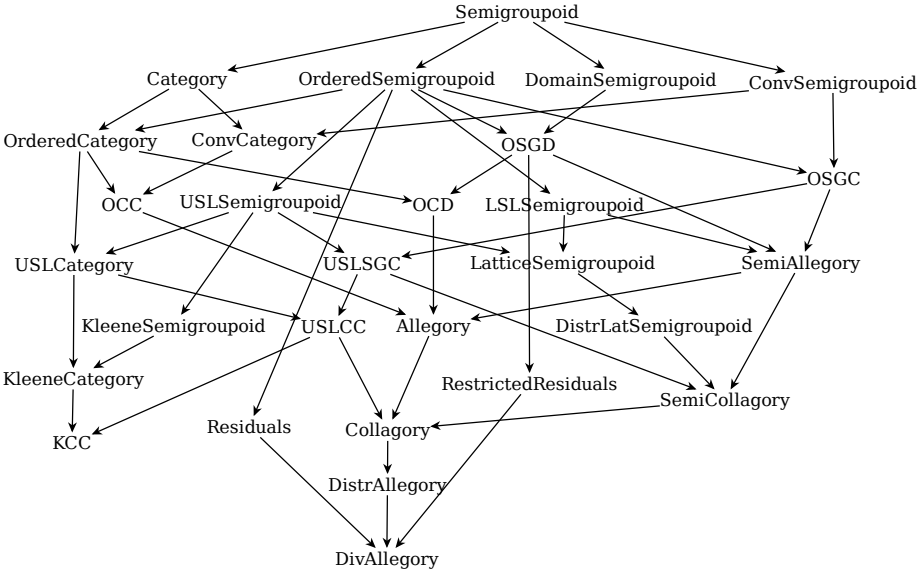
Definition 4.1 A **semigroupoid** $(\mathsf{Obj}, \mathsf{Mor}, \mathsf{src}, \mathsf{trg}, ;)$ is a graph with a set Obj of *objects* as vertices, a set Mor of *morphisms* as edges, with $\mathsf{src}, \mathsf{trg} : \mathsf{Mor} \to \mathsf{Obj}$ assigning source and target object to each morphism (we write "$f : \mathcal{A} \to \mathcal{B}$" instead of "$f \in \mathsf{Mor}$ and $\mathsf{src}\ f = \mathcal{A}$ and $\mathsf{trg}\ f = \mathcal{B}$"), and an additional partial operation "$\_ ; \_$" of composition such that the following hold:

- For $f : \mathcal{A} \to \mathcal{B}$ and $g : \mathcal{B}' \to \mathcal{C}$, the composition $f ; g$ is defined iff $\mathcal{B} = \mathcal{B}'$, and if it is defined, then $(f ; g) : \mathcal{A} \to \mathcal{C}$.
- Composition is associative, i.e., if one of $(f ; g) ; h$ and $f ; (g ; h)$ is defined, then so is the other and they are equal.

For two objects $\mathcal{A}$ and $\mathcal{B}$, the collections of morphisms $f : \mathcal{A} \to \mathcal{B}$ is also called the *homset* from $\mathcal{A}$ to $\mathcal{B}$, and written $\mathsf{Hom}(\mathcal{A}, \mathcal{B})$.

A morphism is called an *endomorphism* iff its source and target objects coincide; an endomorphism R is called *idempotent* if $R ; R = R$. □

From semigroupoids, we obtain more specialised theories by adding in particular identities (to obtain categories), converse, domain, and local ordering of homsets, and their combinations. This continues with semi-lattice and lattice properties for the homsets, and various coherence properties; the following is the inclusion graph of most of our current theories, the most important of which will be discussed in more detail in the remainder of this and the next two sections.



Such a fine-grained modularisation automatically comes with some overhead in comparison with larger, monolithic theories as used for example by Gonzalía [12]. It is interesting to see how the namespace management by Agda's module system, which includes nested modules and records considered as modules, minimises that cost, and enables different approaches to achieving essentially the effect of subtyping with extensible records, which are not available.

Using universe-polymorphism throughout our development gives us the flexibility to use both "small" explicitly constructed examples, like the two-object two-morphism category $\bullet \rightrightarrows \bullet$, standard next-level "large" categories like *Set* and *Rel*, and even larger categories. We use variable names i, j, k, k_1, etc. for universe levels.

As our way to deal with homsets (i.e., sets of morphisms from one object to another) we choose to use the standard-library Setoid theory for individual homsets, starting from a function Hom : Obj → Obj → Setoid j k, and derive from this the underlying function Mor : Obj → Obj → Set j exposing the types of the morphisms, and lifting the equality (i.e., the Setoid equivalence) into the global view:

```
module HomSetoid {i j k : Level} {Obj : Set i} (Hom : Obj → Obj → Setoid j k) where
  Mor : Rel Obj j
  Mor = Setoid.Carrier ∘₂ Hom                -- using: f ∘₂ g = λ x y → f (g x y)
  infix 4 _≈_
  _≈_ : {A B : Obj} → Rel (Mor A B) k
  _≈_ {A} {B} = Setoid._≈_ (Hom A B)
```

This HomSetoid module also includes the lifted properties of the equality, since for using the underlying Setoid properties, one would always need to identify the relevant hom-setoid:

```
  ≈-refl   : {A B : Obj} → {R : Mor A B} → R ≈ R
  ≈-refl     {A} {B} = Setoid.refl (Hom A B)
  ≈-sym    : {A B : Obj} → {R S : Mor A B} → R ≈ S → S ≈ R
  ≈-sym      {A} {B} = Setoid.sym (Hom A B)
  ≈-trans  : {A B : Obj} → {Q R S : Mor A B} → Q ≈ R → R ≈ S → Q ≈ S
  ≈-trans    {A} {B} = Setoid.trans (Hom A B)
```

Equivalently, one could have derived the individual setoids from the global morphism equality, which is essentially the path chosen by Gonzalía [12], and closer in spirit to Def. 4.1. However, we believe the approach chosen here, with Hom primitive instead of src and trg, is more natural in the dependently-typed context, where src and trg are hardly ever needed since their results are typically already available once their argument is, since they are part of the argument's type:

```
  src : {A B : Obj} → Mor A B → Obj
  src   {A} {_} _ = A
  trg : {A B : Obj} → Mor A B → Obj
  trg   {_} {B} _ = B
```

Morphism composition is defined in the context of such a HomSetoid. Thanks to universe-polymorphism together with the formulae-as-types view of relations,

the standard-library type Transitive also provides the typing for composition: the composition operator is a proof for higher-level transitivity of Mor. This typing also turns composition into a total function, so that the definedness discussion in Def. 4.1 does not need to be reflected here.

Due to the setoid setup, we need, besides associativity, also demand a congruence property (defined using the standard-library type _Preserves₂_→_→_).

We also show here one of the definitions of derived concepts that we include in this module (many others are suppressed in this short summary): A morphism I is a *left identity* if for all compatible morphisms R we have I ⨟ R ≈ R:

```
record CompOp {i j k : Level} {Obj : Set i} (Hom : Obj → Obj → Setoid j k)
              : Set (i ⊔ j ⊔ k) where
  open HomSetoid Hom
  infixr 9 _⨟_
  field    _⨟_ : Transitive Mor
           ⨟-cong : {A B C : Obj}
                  → ((_⨟_ {A} {B} {C}) Preserves₂ _≈_ → _≈_ → _≈_)
           ⨟-assoc : {A B C D} {f : Mor A B} {g : Mor B C} {h : Mor C D}
                  → ((f ⨟ g) ⨟ h) ≈ (f ⨟ (g ⨟ h))
  isLeftIdentity : {A : Obj} → Mor A A → Set (i ⊔ j ⊔ k)
  isLeftIdentity {A} I = {B : Obj} {R : Mor A B} → I ⨟ R ≈ R
```

A *semigroupoid* is fully defined by the parameters and fields of a CompOp, and we represent it as a dependent (record) product — the type of the second field depends on the first field Hom.

```
record Semigroupoid {i : Level} (j k : Level) (Obj : Set i) : Set (i ⊔ suc (j ⊔ k)) where
  field Hom    : Obj → Obj → Setoid j k
        compOp : CompOp Hom
  open HomSetoid i j k Hom public
  open CompOp compOp public
```

The last two **open** ... **public** lines serve to re-export everything defined in the (record) modules HomSetoid and CompOp. Since these re-exports do not contribute to the essence of the formalisation, we suppress their rendering from now on. However, these re-exports are essential in that they effectively hide the fact that we modularised the definition of Semigroupoid — a future "**open** Semigroupoid" brings also all the items defined in HomSetoid and CompOp into scope.

Following relation-algebraic terminology, we call the involution of self-dual semigroupoids "*converse*", and define it again in an independent building block ConvOp, including the converse operator and its axioms as **field**s. Combining a Semigroupoid with a ConvOp then produces a ConvSemigroupoid (not shown).

For illustrating the flavour of these developments, we also show here one immediate consequence of the axioms (un˘-cong), the derived concept of symmetry, and a proof that left identities are symmetric. Both proofs are presented in the calculational style, using the mixfix operators ≈-begin_, _≈⟨_⟩_ and _□ which are variants of the calculational reasoning operators provided by the standard library, equipped with two additional implicit object parameters (similar to ≈-sym

etc.) to enable calculational reasoning in homsets without having to specify the homset explicitly.

```
record ConvOp {i j k : Level} {Obj : Set i}
               (SG : Semigroupoid {i} {j} {k} Obj) : Set (i ⊎ j ⊎ k) where
  open Semigroupoid SG
  field _˘            : {A B : Obj} → Mor A B → Mor B A
        ˘-cong        : {A B : Obj}   {R S : Mor A B} → R ≈ S → R ˘ ≈ S ˘
        ˘˘            : {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 ˘)
  un˘-cong : {A B : Obj} {R S : Mor A B} → R ˘ ≈ S ˘ → R ≈ S
  un˘-cong {A} {B} {R} {S} R˘≈S˘ = ≈-begin
    R   ≈⟨ ≈-sym ˘˘ ⟩          (R ˘) ˘
        ≈⟨ ˘-cong R˘≈S˘ ⟩      (S ˘) ˘
        ≈⟨ ˘˘ ⟩                S                                  □
  isSymmetric : {A : Obj} → Mor A A → Set k
  isSymmetric R = R ˘ ≈ R
  isLeftIdentity-isSymmetric : {A : Obj} {R : Mor A A}
                             → isLeftIdentity R → isSymmetric R
  isLeftIdentity-isSymmetric {A} {R} left = ≈-begin
    R ˘ ≈⟨ ≈-sym left ⟩          R ⨟ R ˘
        ≈⟨ ⨟-cong₁ (≈-sym ˘˘) ⟩ (R ˘) ˘ ⨟ R ˘
        ≈⟨ ≈-sym ˘-involution ⟩ (R ⨟ R ˘) ˘
        ≈⟨ ˘-cong left ⟩         (R ˘) ˘
        ≈⟨ ˘˘ ⟩                  R                                 □
```

For locally ordered semigroupoids, we replace the Setoid in the result type of Hom with the stronger Poset, and therefore need to adapt this in the instantiations of HomSetoid and CompOp. The local poset ordering relations are again collected into a global parameterised relation, ⊑. The following is a flattened presentation of the OrderedSemigroupoid definition — the original is composed from several modules and records:

```
record OrderedSemigroupoid {i : Level} (j k₁ k₂ : Level) (Obj : Set i)
                           : Set (i ⊎ suc (j ⊎ k₁ ⊎ k₂)) where
  field Hom : Obj → Obj → Poset j k₁ k₂
        compOp : CompOp i j k₁ (posetSetoid ∘₂ Hom)
  semigroupoid : Semigroupoid Obj
  semigroupoid = record {Hom = posetSetoid ∘₂ Hom; compOp = compOp}
  open Semigroupoid semigroupoid hiding (Hom; compOp)
  infix 4 _⊑_
  _⊑_ : {A B : Obj} → Rel (Mor A B) k₂   -- The morphism ordering.
  _⊑_ {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')
```

The definition and **open**ing of semigroupoid here produces the illusion that an OrderedSemigroupoid is a "Semigroupoid with a local order on each homset", even

though the definition has been structured in a different way. From now on we will suppress the definitions of such subtheories, like semigroupoid here, if they are not used inside the shown part of the enclosing definition.

Also included in OrderedSemigroupoid (but not shown) are proofs of the ordering properties expressed in terms of $\sqsubseteq$, one-sided monotonicity properties of composition, and numerous derived concepts and laws, including properties and types of idempotent subidentities.

An *ordered semigroupoid with converse (OSGC)* adds to its constituents the monotonicity law for converse (and contains a large number of derived concepts and lemmas, not shown here):

```
record OSGC {i : Level} (j k₁ k₂ : Level) (Obj : Set i) : Set (i ⊔ suc (j ⊔ k₁ ⊔ k₂)) where
  field orderedSemigroupoid : OrderedSemigroupoid j k₁ k₂ Obj
  open OrderedSemigroupoid orderedSemigroupoid

  field convOp : ConvOp semigroupoid
  open ConvOp convOp

  field ˘-monotone : {A B : Obj} {R S : Mor A B} → R ⊑ S → (R ˘) ⊑ (S ˘)
```

When defining *upper* and *lower semilattice semigroupoids*, we do not add the join respectively meet operators directly to the range of Hom, since the standard library currently only provides lattices, but not semilattices. Otherwise, these definitions are straightforward and not shown.

Semi-allegories are, by analogy with semigroupoids, "allegories without identity morphisms", i.e., lower semilattice semigroupoids with converse and domain (see Sect. 5) satisfying the Dedekind rule:

```
record SemiAllegory {i : Level} (j k₁ k₂ : Level) (Obj : Set i)
                    : Set (i ⊔ suc (j ⊔ k₁ ⊔ k₂)) where
  field osgc          : OSGC j k₁ k₂ Obj
  open OSGC           osgc
  field meetOp        : MeetOp        orderedSemigroupoid
        domainOp      : OSGDomainOp orderedSemigroupoid
  open MeetOp         meetOp
  open OSGDomainOp domainOp

  field Dedekind : {A B C : Obj} {Q : Mor A B} {R : Mor B C} {S : Mor A C}
                   → (Q ⨾ R ⊓ S) ⊑ (Q ⊓ S ⨾ R ˘) ⨾ (R ⊓ Q ˘ ⨾ S)
```

We now turn to categories, where the new ingredient is the operation assigning an identity morphism to each object:

```
record IdOp (i j k : Level) {Obj : Set i} (Hom : Obj → Obj → Setoid j k)
            (_⨾_ : Transitive (Setoid.Carrier ∘₂ Hom)) : Set (i ⊔ j ⊔ k) where
  open HomSetoid i j k Hom

  field Id      : (A   : Obj) → Mor A A
        leftId  : {A B : Obj} → {f : Mor A B} → (Id A ⨾ f) ≈ f
        rightId : {A B : Obj} → {f : Mor A B} → (f ⨾ Id B) ≈ f
```

This identity module can now be added easily to (ordered) semigroupoids to produce the corresponding categories:

– A Category consists of a Semigroupoid and an IdOp
– An OrderedCategory consists of an OrderedSemigroupoid and an IdOp.
– A ConvCategory consists of a ConvSemigroupoid and an IdOp; preservation of identities by converse follows from the lemma shown in ConvSemigroupoid:

```
Id˘ : {A : Obj} → (Id {A}) ˘ ≈ Id {A}
Id˘ {A} = isLeftIdentity-isSymmetric leftId
```

– An OCC (*ordered category with converse*) consists of an OSGC and an IdOp.
– An Allegory is an OCC with a MeetOp satisfying Dedekind, and can be shown to contain a SemiAllegory by defining a DomainOp based on:

```
dom : {A B : Obj} → Mor A B → Mor A A
dom R = Id ⊓ R ⨾ R ˘
```

  Since the OCC theory exports so much material, it appears more natural to use OCCs as starting point for defining allegories analogously to semiallegories — the alternative would have been to start from semiallegories and work analogously to the OCC definition, which in this case would have required careful re-export of the OCC material since Agda currently does not permit re-export of the same item via more than one interface.
– Similarly, a (Semi-)Collagory is a (Semi-)Allegory with a JoinOp and lattice distributivity.
– A DistrAllegory is a Collagory with least morphisms and zero laws.

In OCCs, we also have the standard relation-algebraic way of defining properties like univalence ((R ˘ ⨾ R) ⊑ Id B), totality (Id A ⊑ (R ⨾ R ˘)), injectivity, etc., and deriving laws for them. However, since most reasoning with these properties immediately uses the identity laws of composition, the corresponding definitions in OSGC typically make for shorter proofs, and we use those to define a proof-carrying type of Mappings (all in OSGC):

```
isUnivalent : {A B : Obj} → Mor A B → Set (i ⊔ j ⊔ k₂)
isUnivalent R = isSubidentity (R ˘ ⨾ R)

isTotal : {A B : Obj} → Mor A B → Set (i ⊔ j ⊔ k₂)
isTotal R = isSuperidentity (R ⨾ R ˘)

isMapping : {A B : Obj} → Mor A B → Set (i ⊔ j ⊔ k₂)
isMapping R = isUnivalent R × isTotal R

record Mapping (A B : Obj) : Set (i ⊔ j ⊔ k₂) where field mor : Mor A B
                                                         prf : isMapping mor
```

The semigroupoid of mappings in an OSGC and the category of mappings in an OCC are easily constructed (in module Categoric.MapSG):

```
MapSG : {i j k₁ k₂ : Level} {Obj : Set i}
      → OSGC j k₁ k₂ Obj → Semigroupoid (i ⊔ j ⊔ k₂) k₁ Obj
MapCat : {i j k₁ k₂ : Level} {Obj : Set i}
      → OCC j k₁ k₂ Obj → Category (i ⊔ j ⊔ k₂) k₁ Obj
```

5 Domain

Domain can be defined in allegories as shown above; for weaker theories, Desharnais *et al.* [9] axiomatised domain operators essentially in an ordering context (in semirings and Kleene algebras), where the domain operator produces subidentities. A more recent alternative is the purely equational approach of Desharnais *et al.* [7], which starts just from semigroups. We have formalised both approaches, concentrating on the aspects that do not require complements.

Most of [7, Section 3] has been generalised to *left closure semigroupoids*:

```
field
  dom : {A B : Obj} → Mor A B → Mor A A
  dom-cong : {A B : Obj} {R S : Mor A B} → R ≈ S → dom R ≈ dom S
  D1 : {A B : Obj} {R : Mor A B} → (dom R) ⨟ R ≈ R
  L2 : {A B : Obj} {R : Mor A B} → dom (dom R) ≈ dom R
  L3 : {A B C} {R : Mor A B} {S : Mor B C} → (dom R) ⨟ dom (R ⨟ S) ≈ dom (R ⨟ S)
  D4 : {A B C} {R : Mor A B} {S : Mor A C}
     → (dom R) ⨟ (dom S) ≈ (dom S) ⨟ (dom R)

_≼_ : {A B : Obj} (R S : Mor A B) → Set k                -- The "fundamental order"
R ≼ S = R ≈ (dom R) ⨟ S
```

For the fundamental order $\preccurlyeq$, we have been able to show some additional properties, namely that it is preserved by multiplication with domain elements from the left, and that the domain semigroup axiom D3 implies monotonicity of dom with respect to $\preccurlyeq$ (proofs not shown):

```
dom⨟-≼monotone    : {A B C : Obj} {Q : Mor A B} {R S : Mor A C}
                  → R ≼ S → ((dom Q) ⨟ R) ≼ ((dom Q) ⨟ S)
dom-D3-≼monotone : {A B : Obj} {R S : Mor A B}
          → (dom ((dom R) ⨟ S) ≈ (dom R) ⨟ (dom S)) → R ≼ S → dom R ≼ dom S
```

However, $\preccurlyeq$-monotonicity of dom does not imply D3; the model searcher Mace4 [24] finds a four-element counter-example.

For the subidentity-based approach, we adapt the definitions of [9] to the ordered semigroupoid setting:

```
record OSGDomainOp {i j k₁ k₂ : Level} {Obj : Set i}
       (base : OrderedSemigroupoid j k₁ k₂ Obj) : Set (i ⊔ j ⊔ k₁ ⊔ k₂) where
  open OrderedSemigroupoid base
  field dom : {A B : Obj} → Mor A B → Mor A A
        domSubIdentity     : {A B : Obj} {R : Mor A B} → isSubidentity (dom R)
        dom-⨟-idempotent   : {A B : Obj} {R : Mor A B} → (dom R) ⨟ (dom R) ≈ dom R
        domPreserves⊑      : {A B : Obj} {Q R : Mor A B} → Q ⊑ R → Q ⊑ (dom R) ⨟ Q
        domLeastPreserver : {A B : Obj} {R : Mor A B} {d : Mor A A}
                          → isSubidentity d → (d ⨟ d ≈ d) → (R ⊑ d ⨟ R) → dom R ⊑ d
        domLocality        : {A B C : Obj} {R : Mor A B} {S : Mor B C}
                          → dom (R ⨟ dom S) ⊑ dom (R ⨟ S)
```

Here, we show that this satisfies all the conditions of a domain semigroupoid, and define *domain minimality*, which has been proposed by Desharnais and Möller [8] for characterising determinacy in Kleene algebras.

6 Restricted Residuals

Motivated by application to relations between infinite sets, where residuals (with respect to composition) of finite relations typically have an infinite "uninteresting part", [19] introduced *restricted residuals* that characterise the finite "interesting part"; they have also found applications to substitutions [22]. For ordered semigroupoids with domain dom and range ran, restricted residuals are distinguished from standard residuals by the additional restr axiom:

$$
\begin{array}{lll}
\textbf{field}\ \_\not\backslash\_ & : \{A\ B\ C\} \to \mathsf{Mor}\ A\ C \to \mathsf{Mor}\ B\ C \to \mathsf{Mor}\ A\ B \\
\not\backslash\text{-cancel-outer} & : \{A\ B\ C\}\ \{S : \mathsf{Mor}\ A\ C\}\ \{R : \mathsf{Mor}\ B\ C\} \to (S \not\backslash R) \fatsemi R \sqsubseteq S \\
\not\backslash\text{-restr} & : \{A\ B\ C\}\ \{S : \mathsf{Mor}\ A\ C\}\ \{R : \mathsf{Mor}\ B\ C\} \to \mathsf{ran}\ (S \not\backslash R) \sqsubseteq \mathsf{dom}\ R \\
\not\backslash\text{-universal} & : \{A\ B\ C\}\ \{S : \mathsf{Mor}\ A\ C\}\ \{R : \mathsf{Mor}\ B\ C\}\ \{Q : \mathsf{Mor}\ A\ B\} \\
 & \to Q \fatsemi R \sqsubseteq S \to \mathsf{ran}\ Q \sqsubseteq \mathsf{dom}\ R \to Q \sqsubseteq S \not\backslash R
\end{array}
$$

From the many properties of standard residuals (see for example [11]), a remarkable number carries over to restricted residuals; we list these derived properties without their proofs and without their implicit arguments:

$$
\begin{array}{lll}
\not\backslash\text{-cancel-inner} & : \mathsf{ran}\ T \sqsubseteq \mathsf{dom}\ S & \to T \sqsubseteq (T \fatsemi S) \not\backslash S \\
\not\backslash\text{-monotone} & : S_1 \sqsubseteq S_2 & \to S_1 \not\backslash R \sqsubseteq S_2 \not\backslash R \\
\not\backslash\text{-antitone} & : R_2 \sqsubseteq R_1 \to \mathsf{dom}\ R_1 \sqsubseteq \mathsf{dom}\ R_2 & \to S \not\backslash R_1 \sqsubseteq S \not\backslash R_2 \\
\not\backslash\text{-cancel-middle} & : & (S \not\backslash R) \fatsemi (R \not\backslash T) \sqsubseteq S \not\backslash T \\
\not\backslash\text{-cancel-}\fatsemi & : \mathsf{ran}\ (S \not\backslash R) \sqsubseteq \mathsf{dom}\ (R \fatsemi T) & \to S \not\backslash R \sqsubseteq (S \fatsemi T) \not\backslash (R \fatsemi T) \\
\not\backslash\text{-outer-}\fatsemi & : & F \fatsemi (S \not\backslash R) \sqsubseteq (F \fatsemi S) \not\backslash R \\
\mathsf{dom}\text{-}\not\backslash & : & \mathsf{dom}\ (S \not\backslash R) \sqsubseteq \mathsf{dom}\ S \\
\mathsf{domS}\not\backslash\mathsf{S}\approx\mathsf{domS} & : & \mathsf{dom}\ (S \not\backslash S) \approx \mathsf{dom}\ S \\
\mathsf{ranS}\not\backslash\mathsf{S}\approx\mathsf{domS} & : & \mathsf{ran}\ (S \not\backslash S) \approx \mathsf{dom}\ S \\
\mathsf{S}\not\backslash\mathsf{S}\text{-}\fatsemi\text{-}\mathsf{S} & : & (S \not\backslash S) \fatsemi S \approx S \\
\mathsf{S}\not\backslash\mathsf{S}\text{-isTransitive} & : & \mathsf{isTransitive}\ (S \not\backslash S)
\end{array}
$$

(The property $\not\backslash$-cancel-middle has first been shown in [15].) Restricted right residuals are defined dually, and the following laws hold for combining the two:

$$
\begin{array}{lll}
\not/\text{-twist} & : \mathsf{dom}\ (S \not/ R) \sqsubseteq \mathsf{ran}\ (T \not/ S) \to S \not/ R \sqsubseteq (T \not/ S) \not\backslash (T \not/ R) \\
\not/\text{-twist-down} & : \mathsf{dom}\ (S \not/ R) \sqsubseteq \mathsf{ran}\ (R \not/ S) \to S \not/ R \sqsubseteq (R \not/ S) \not\backslash (R \not/ R) \\
\not/\text{-twist-up} & : & S \not/ R \sqsubseteq (S \not/ S) \not\backslash (S \not/ R)
\end{array}
$$

7 Generalised Algebras

In the context of many-sorted algebras, a signature consists of a set of sorts and a set of function symbols, each equipped with information about its argument and result sorts. An algebra consists of interpretations of the syntactic elements of its signature. Typically, sorts are interpreted as sets, and function symbols as functions from the Cartesian products of the argument sort interpretations to the target sort interpretation.

For a signature Σ, the type of Σ-algebras is then not a "small" Set, but a "large" Set_1. i.e., a member of the next universe encompassing Set. essentially since there

are at least as many Σ-algebras as there are "small" Sets. This implies that we cannot, at the same universe level, directly define an algebra which has a carrier that is some set of algebras of some possibly different signature. A practical example where this is desirable is algebras of graphs with graph operations, where the graphs themselves are considered as unary algebras. Therefore, we need a universe-polymorphic concept of Σ-algebras.

In addition to universe polymorphism, we also require *shape polymorphism*, where a shape specifies the allowed arities of function symbols. The resulting allegories of algebras satisfy different properties depending on these shapes (as indicated):

```
data SHAPE : Set where LIST   : SHAPE  -- arbitrary arities: Allegory
                       MAYBE  : SHAPE  -- only 0-ary and unary: Collagory
                       NELIST : SHAPE  -- not 0-ary: Allegory with zero laws
                       ONE    : SHAPE  -- only unary: Distributive allegory
```

We introduce auxiliary definitions ShapedList, ShapeProduct and ShapeFunctor etc. to deal with shape polymorphism, and define FunSig as type for the signatures of single function symbols, where src is the argument sort list and has its possible lengths determined by Shape:

```
record FunSig (Shape : SHAPE) (Sort : Set) : Set where
  field src : ShapedList Shape Sort
        trg : Sort
```

A signature of a given Shape and with Sort as set of sorts and FSymb as set of function symbols provides for each function symbol an individual signature (FunSig):

```
Sig : (Shape : SHAPE) → (Sort : Set) → (FSymb : Set) → Set
Sig Shape Sort FSymb = FSymb → FunSig Shape Sort
```

Furthermore, we also need *category polymorphism*, since we want to be able to interpret our signatures over categories different from *Set* — in the graph operation example mentioned above, we might want to move to some category of graph homomorphisms.

For defining just an algebra, we need neither semigroupoid composition nor ShapeProduct parallel composition of morphisms, and therefore declare only the necessary parameters, which include the mapping objProd that takes a Shaped list of semigroupoid objects and maps them to the corresponding product object:

```
record Algebra {Shape : SHAPE} {Sort FSymb} (sig : Sig Shape Sort FSymb)
  {i j : Level} (Obj : Set i) (Mor : Rel Obj j) (objProd : ShapedList Shape Obj → Obj)
  : Set (i ⊔ j) where
  field carrier : Sort → Obj   -- interpretation of sorts
        op : (f : FSymb)       -- interpretation of function symbols
           → Mor (objProd (mapShapedList Shape carrier (FunSig.src (sig f))))
                 (                                     carrier (FunSig.trg (sig f)))
```

Declaring **open** Algebra makes the field selectors carrier and op available unqualified. This allows a quite concise definition of bisimulations, or relational algebra homomorphisms, where the morphisms used as function symbol interpretations are restricted to be mappings in an OSGC. Such a bisimulation is a sort-indexed family of morphisms between the respective interpretations of each sort, together with a proof of the bisimulation property:

```
record AlgBiSim
  {Shape : SHAPE} {Sort : Set} {FSymb : Set}
  (sig : Sig Shape Sort FSymb)
  {i j k₁ k₂ : Level} {Obj : Set i}
  (base : OSGC j k₁ k₂ Obj)
  (P : ShapeProductSGFunctor Shape (OSGC.semigroupoid base))
  (A : Algebra sig Obj (OSGC.Mapping base) (ShapeProductSGFunctor.objProd P))
  (B : Algebra sig Obj (OSGC.Mapping base) (ShapeProductSGFunctor.objProd P))
  : Set (i ⊔ j ⊔ k₁ ⊔ k₂)
  where
    open OSGC base
    open ShapeProductSGFunctor P
    field
      hom      : (s : Sort) → Mor (carrier A s) (carrier B s)
      commutes : (f : FSymb)
                 → morProd hom (FunSig.src (sig f)) ⨾ Mapping.mor (op B f)
                 ⊑ Mapping.mor (op A f) ⨾ hom (FunSig.trg (sig f))
```

As a conventional mathematical definition, this might be expressed as follows:

Definition 7.1 Let a signature $\Sigma = (\mathcal{S}, \mathcal{F}, \mathsf{src}, \mathsf{trg})$, an OSGC $\mathbf{C}$ with sufficient direct products, and two abstract Σ-algebras $\mathcal{A}$ and $\mathcal{B}$ over $\mathbf{C}$ be given.

A *Σ-bisimulation Φ from $\mathcal{A}$ to $\mathcal{B}$* is an $\mathcal{S}$-indexed family of $\mathbf{C}$-morphisms $\Phi_s : s^{\mathcal{A}} \to s^{\mathcal{B}}$ such that for every function symbol $f \in \mathcal{F}$ with $f : s_1 \times \cdots \times s_n \to t$ the following inclusion holds:

$$(\Phi_{s_1} \times \cdots \times \Phi_{s_n}) \mathbin{;} f^{\mathcal{B}} \sqsubseteq f^{\mathcal{A}} \mathbin{;} \Phi_t \quad . \qquad \square$$

As usual in such conventional mathematics, many parameters are left implicit, and even where they are technically turned into implicit parameters in Agda, they need to be explicitly listed in the definition. For example, in Def. 7.1 the Shape of the signature is not mentioned explicitly at all, but, together with the ShapeProductSGFunctor, subsumed in the phrase "with sufficient direct products".

Composition of bisimulations is component-wise composition of the morphism families; the necessary correctness proof is, due to explicit associativity steps etc., a bit longer than in usual mathematical presentations, but quite readable — prodComp is distributivity of the morphism part morProd of the Shaped list product functor P over composition:

```
let homComp = λ s → hom R s ⨾ hom S s in record
  {hom = homComp
  ;commutes = λ f → let open FunSig (sig f) using (src; trg) in ⊑-begin
      morProd homComp src ⨾ Mapping.mor (op C f)
```

```
≈⟨ ⨟-cong₁ prodComp ⟩
  (morProd (hom R) src ⨟ morProd (hom S) src) ⨟ Mapping.mor (op C f)
≈⟨ ⨟-assoc ⟩
  morProd (hom R) src ⨟ morProd (hom S) src ⨟ Mapping.mor (op C f)
⊑⟨ ⨟-monotone₂ (commutes S f) ⟩
  morProd (hom R) src ⨟ Mapping.mor (op B f) ⨟ hom S trg
≈⟨ ⨟-assocL ⟩
  (morProd (hom R) src ⨟ Mapping.mor (op B f)) ⨟ hom S trg
⊑⟨ ⨟-monotone₁ (commutes R f) ⟩
  (Mapping.mor (op A f) ⨟ hom R trg) ⨟ hom S trg
≈⟨ ⨟-assoc ⟩
  Mapping.mor (op A f) ⨟ hom R trg ⨟ hom S trg
≈⟨ ≈-refl ⟩
  Mapping.mor (op A f) ⨟ homComp trg                    □}
```

From there, it is relatively straightforward to define the instances of the semigroupoid and category types introduced in Sect. 4.

8 Related Work

Our approach to categories with setoids of morphisms, but not of objects, derives essentially from Kanda's "effective categories" [23]; it is also used by Huet and Saïbi [16] for their formalisation of category theory in Coq, and by Gonzalía [12], who produced formalisations of concrete heterogeneous binary relations and of Freyd and Scedrov's allegory hierarchy [10] in Alf, a predecessor of Agda.

Mu *et al.* [25] have contributed Agda2 theories inspired by Bird and de Moor's *Algebra of Programming* [4]; they note the advantages that Agda2 brought to their formalisations of concrete relations over the Alf formalisations of Gonzalía.

Jackson [17] formalised abstract algebra as used in computer algebra systems in Nuprl, which uses a variant of type theory that provides sets, and therefore does not need setoids. This work also does not include a general approach using signatures.

Capretta [5] formalised universal algebra in Coq, with fixed encoding of the sets of sorts and function symbols as finite natural number sets. Barthe *et al.* [2] provide an in-depth discussion of different treatments of setoids.

9 Conclusion

Our extension of a treatment of binary heterogeneous relations similar to that of Mu *et al.* [25] to full universe polymorphism as mentioned in Sect. 3 is a minor, technical contribution which nonetheless constitutes a significant generalisation.

The semigroupoids and categories of Sect. 4 not only bring formalisations similar to Gonzalía's [12] into a current system, and into fully universe-polymorphic shape; they also reflect recent developments towards finer granularity of these theories. They are also a significant advance over the Isabelle theories of [18] both in scope and in style of exposition: Besides of the more natural formalisation of categories in a dependently-typed system, Agda also enables more

flexibility with structuring a theory hierarchy through arbitrary (sub-)module **open**ing, whereas the records underlying locales in Isabelle allow only extension at predefined extension points. As a result, the Agda formalisation appears to be much more maintainable.

Finally, our way of constructing allegories (etc.) of algebras from underlying allegories seems to not have been formalised in a mechanised theorem prover before. Doing this in Agda has proven quite satisfactory, since the language combines natural mathematical expressiveness with a programming attitude. Future work will continue to formalise material required for powerful relation-algebraic graph transformation concepts [21], and will explore to use Agda's foreign-function interface to Haskell to combine verified graph transformation algorithms in Agda with graphical user interfaces [29], or to use it in code generation back-ends [1].

References

1. Anand, C.K., Kahl, W.: An optimized Cell BE special function library generated by Coconut. IEEE Transactions on Computers 58(8), 1126–1138 (2009)
2. Barthe, G., Capretta, V., Pons, O.: Setoids in type theory. J. Funct. Program. 13(2), 261–293 (2003)
3. Berghammer, R., Jaoua, A.M., Möller, B. (eds.): RelMiCS 2009. LNCS, vol. 5827. Springer, Heidelberg (2009)
4. Bird, R.S., de Moor, O.: Algebra of Programming. International Series in Computer Science, vol. 100. Prentice-Hall, Englewood Cliffs (1997)
5. Capretta, V.: Universal algebra in type theory. In: Bertot, Y., Dowek, G., Hirschowitz, A., Paulin, C., Théry, L. (eds.) TPHOLs 1999. LNCS, vol. 1690, pp. 131–148. Springer, Heidelberg (1999)
6. Corradini, A., Montanari, U., Rossi, F., Ehrig, H., Heckel, R., Löwe, M.: Algebraic approaches to graph transformation, part I: Basic concepts and double pushout approach. In: Rozenberg, G. (ed.) Handbook of Graph Grammars and Computing by Graph Transformation, Foundations, vol. 1, ch. 3, pp. 163–245. World Scientific, Singapore (1997)
7. Desharnais, J., Jipsen, P., Struth, G.: Domain and antidomain semigroups. In: Berghammer et al. [3], pp. 73–87
8. Desharnais, J., Möller, B.: Characterizing determinacy in Kleene algebras. Information Sciences 139, 253–273 (2001)
9. Desharnais, J., Möller, B., Struth, G.: Kleene algebra with domain. ACM Transactions on Computational Logic 7(4), 798–833 (2006)
10. Freyd, P.J., Scedrov, A.: Categories, Allegories. North-Holland Mathematical Library, vol. 39. North-Holland, Amsterdam (1990)
11. Furusawa, H., Kahl, W.: A study on symmetric quotients. Tech. Rep. 1998-06, Fakultät für Informatik, Universität der Bundeswehr München (December 1998)
12. Gonzalía, C.: Relations in Dependent Type Theory. Ph.D. thesis, also as Technical Report No. 14D, Department of Computer Science and Engineering, Chalmers University of Technology, Göteborg University (2006)
13. Gurevich, Y.: Evolving Algebras: An attempt to discover semantics. In: Rozenberg, G., Salomaa, A. (eds.) Current Trends in Theoretical Computer Science, pp. 266–292. World Scientific, Singapore (1993)

14. Gurevich, Y.: Sequential abstract state machines capture sequential algorithms. ACM Transactions on Computational Logic 1(1), 77–111 (2000)
15. Han, J.: Proofs of Relational Semigroupoids in Isabelle/Isar. M.Sc. thesis, McMaster University, Department of Computing and Software (2008)
16. Huet, G., Saïbi, A.: Constructive category theory. In: Plotkin, G.D., Stirling, C., Tofte, M. (eds.) Proof, language, and interaction: Essays in honour of Robin Milner. Foundations of Computing Series, pp. 239–275. MIT Press, Cambridge (2000)
17. Jackson, P.B.: Enhancing the Nuprl Proof Development System and Applying it to Computational Abstract Algebra. Ph.D. thesis, Cornell University (1995)
18. Kahl, W.: Calculational relation-algebraic proofs in Isabelle/Isar. In: Berghammer, R., Möller, B., Struth, G. (eds.) RelMiCS 2003. LNCS, vol. 3051, pp. 178–190. Springer, Heidelberg (2004)
19. Kahl, W.: Relational semigroupoids: Abstract relation-algebraic interfaces for finite relations between infinite types. J. Logic and Algebraic Programming 76(1), 60–89 (2008)
20. Kahl, W.: Collagories for relational adhesive rewriting. In: Berghammer et al [3], pp. 211–226
21. Kahl, W.: Amalgamating pushout and pullback graph transformation in collagories. In: Ehrig, H., Rensink, A., Rozenberg, G., Schürr, A. (eds.) ICGT 2010. LNCS, vol. 6372, pp. 362–378. Springer, Heidelberg (2010)
22. Kahl, W.: Determinisation of relational substitutions in ordered categories with domain. J. Logic and Algebraic Programming 79, 812–829 (2010)
23. Kanda, A.: Constructive category theory (no. 1). In: Gruska, J., Chytil, M.P. (eds.) MFCS 1981. LNCS, vol. 118, pp. 563–577. Springer, Heidelberg (1981)
24. McCune, W.: Prover9 and Mace4, version LADR-2009-11A (2009), `http://www.prover9.org/`
25. Mu, S.C., Ko, H.S., Jansson, P.: Algebra of programming using dependent types. In: Audebaud, P., Paulin-Mohring, C. (eds.) MPC 2008. LNCS, vol. 5133, pp. 268–283. Springer, Heidelberg (2008)
26. Norell, U.: Towards a Practical Programming Language Based on Dependent Type Theory. Ph.D. thesis, Department of Computer Science and Engineering, Chalmers University of Technology (September 2007)
27. Schmidt, G., Hattensperger, C., Winter, M.: Heterogeneous relation algebra. In: Brink, C., Kahl, W., Schmidt, G. (eds.) Relational Methods in Computer Science. Advances in Computing Science, ch. 3, pp. 39–53. Springer, Wien (1997)
28. Schmidt, G., Ströhlein, T.: Relations and Graphs, Discrete Mathematics for Computer Scientists. EATCS-Monographs on Theoret. Comput. Sci. Springer, Heidelberg (1993)
29. West, S., Kahl, W.: A generic graph transformation, visualisation, and editing framework in Haskell. In: Boronat, A., Heckel, R. (eds.) Proceedings of the Eighth International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2009). Electronic Communications of the EASST, vol. 18, pp. 12.1–12.18 (September 2009)