Exploring (Finite) Relation Algebras Using Tools Written in Haskell

WOLFRAM KAHL

GUNTHER SCHMIDT

Bericht Nr. 2000-02

Oktober 2000

Universität der Bundeswehr München

Fakultät für

INFORMATIK

Werner-Heisenberg-Weg 39 • 85577 Neubiberg • Germany

Exploring (Finite) Relation Algebras Using Tools Written in Haskell

Wolfram Kahl　　　　Gunther Schmidt

Institute for Software Technology
Department of Computing Science
Federal Armed Forces University Munich
e-Mail: `{Kahl|Schmidt}@Informatik.UniBw-Muenchen.DE`

31 October 2000

Abstract

During the last few years, relational methods have been gaining more and more acceptance and impact in computer science. Besides applications of concrete relations, also non-standard models of the relation algebraic axioms are important in fields as far apart as artificial intelligence and distributed computing. Also weaker structures have been considered, such as Dedekind categories in connection with fuzzy reasoning, and different kinds of allegories.
In this report we present a library of Haskell modules that allows to explore relation algebras and several weaker structures by providing different means to construct and test such algebras.
The kernel of our library is strictly conformant to the Haskell 98 standard, and can therefore be expected to be usable on future Haskell systems, too. For ease of use, we additionally provide a more elegant interface using non-standard extensions.

Contents

Introduction

All of us are accustomed to a bit of reasoning with relations such as *is greater than*, *is equal to*, *is the brother of*, *is the father of*, etc. The mechanics of such reasoning have long been traced back to their algebraic laws, yielding the concept of (heterogeneous) relation algebra. In addition it has been shown that suitable products, sub-algebras and matrix algebras with coefficients taken from given relation algebras are relation algebras again.

Often, also slightly weaker structures are studied such as allegories, distributive allegories, division allegories, and Dedekind categories.

Here, a common framework is presented for calculational work with all the structures mentioned. It takes into account that they share concepts and properties so as to be able to, e.g., introduce the idea of division only once for division allegories and to directly reuse it for the more specific Dedekind allegories as well as for relation algebras. Chapter 1 is mainly devoted to the presentation of a Haskell program in literate style to administer any given structure of the kinds mentioned and to scrupulously test for all mathematical properties such structures should fulfil to be well-defined. The underlying source of the whole report constitutes executable Haskell code and is available from the RATH home page:

URL: http://ist.unibw-muenchen.de/relmics/tools/RATH/

The classical model of abstract relation algebra is given by all the relations on a set or between sets. For the tools presented in Chapter 2, this is just one specific case. The tools are also designed to handle product algebras, sub-algebras, and matrix algebras on or over relation algebras.

As often experienced in other application fields, however, while going back to the algebraic laws for relations, it turned out that other models one had not thought of so far obeyed the same laws. So one has in addition to the classical ones non-standard models of relation algebra. To cope with these, Chapter 2 provides a toolbox to construct arbitrary relation algebras from atom sets.

To give an impression of possible behaviour of non-standard relation algebras, we present a few examples in Chapter 3, among these there are mereological considerations in spatial reasoning (see Sect. 3.3), interval algebras (see Sect. 3.4), and compass algebras (see Sect. 3.5), abstracting several fields of everyday life. In addition, the McKenzie model is recalled together with the proof that it cannot be represented in an algebra of relations, (see Sect. 3.1) — the first non-representbale relation algebras were found by Roger Lyndon, who published one of them in 1950 [Lyn50]. Another small example gives relation algebras with a surprising property: The product of two universal relations need not itself be a universal relation, (see Sect. 3.6).

The investigation of this diversity of *small* models is justified as these are candidates for being basic blocks of products and matrices to form bigger relation algebras later on.

That relations in the classical sense cannot be given a finite axiom system has been known for a long time. The axiom system presented, therefore, allows the additional models already mentioned. On the other hand side, there is a formula that is obviously satisfied for relations in the classical sense, but has for a very long time not been deduced from the axioms. While proofs have often been tried introducing additional assumptions, e.g. [Des99], here a model is given where this formula does not hold, (see Sect. 1.3.6, Sect. 3.2).

As this formula more or less describes that composition distributes over parallel execution, the presentation of a non-standard model where this formula fails to hold is not uninteresting.

In studying this question, other useful relation algebras have been found that model nonstrict situations. Information on an object that is a pair of two elementary items is now conceived as having 4 possible values with an obvious ordering between them resembling increasing information: Both elements known, left object known while the other is not, right object known while the other is not, none of them is known. It is still possible to handle this case with abstract relation algebra.

A Few Historical Remarks

Relations may not be traced back to Aristotle (384–322 b.C.). Namely, given a horse — which certainly is an animal — we are unable to infer (by the method of syllogism attributed to him) that the head of a horse is the head of an animal. However, much of our topic dates back to 1847, when George Boole started publishing his *The mathematical analysis of logic, being an essay toward a calculus of deductive reasoning* and later articles such as the famous *An investigation on the laws of thought* of 1854. Already in 1859, Augustus De Morgan, in parallel to his inventing of the broadly known rule $\overline{P \vee Q} = \overline{P} \wedge \overline{Q}$ [DM50], proved a so-called “Theorem K” [DM60]. For more than a century, people obviously never read thus far in his papers. If they had, they might have recognised the importance of this theorem. It was only since 1990 that researchers as Roger Maddux seem to have traced modern developments back to this theorem which is the Schröder rule in a different notation; interestingly, De Morgan already seemed to give it an essentially axiomatic rôle.

Around 1870, Charles Sanders Peirce looked for a suitable *Algebra of Logic.* His books and his biography may be found even today in first class book stores oriented towards philosophy and logic.

Later in 1895, Ernst Schröder published his gigantic 3-volume-collection on the algebra of logic [Sch95]. This huge pile of formulae is by no means exhausted today.

One should keep in mind that at that time matrix notation for linear algebra had not yet been developed, or at least was not commonly being used. Matrices seem to have their origin in work on geometry and group theory by Artur Cayley, Hermann Günther Graßmann, August Ferdinand Möbius, and Sir William Rowan Hamilton. Later, Otto Toeplitz worked extensively with matrices. While algebra of logic seemed to be on a good way, more or less in parallel to inventing set theory, the turmoil on set paradoxes and the brilliant performance of Bertrand Russell and Alfred North Whitehead along their *Principia Mathematica* prevented people from working on relations for nearly half a century.

It was the great Alfred Tarski in 1941, who revitalised relation algebra, who educated scholars, and who raised the contemporary interest in this field from the theoretical side. With a paper of 1948, Jacques Riguet studied and collected relations in a way that anticipated many of the applications of today.

With seminars in Schloß Dagstuhl (Germany, January 1994), in Paraty (Rio de Janeiro, August 1995), in Hammamet (Tunisia, January 1997), in Warsaw (September 1998), in Valcartier (Québec, January 2000) followed by a seminar planned for autumn 2001, an international group of scientists has now been formed, meeting regularly in a one-and-a-half year rhythm. So, a much more rapid development may be expected from now on.

Related Work

Computer support for relation algebraic explorations mostly follows one of two approaches: the theorem proving approach and the simulation approach.

For the first approach, let us mention the interactive proof assistant RALF [HBS94, BH94, Hat97, KH98], the Isabelle theory RALL [vOG97]. Also the PhD thesis of Peter Jipsen [Jip92] essentially belongs into this camp, since it employs theorem proving methods to automatically explore candidate algebras.

In the simulation approach, the most well-known system is RelView [ATBS89, BBS97], which allows sophisticated manipulation of concrete relations.

To some extent, the present work might be considered as an attempt to provide a RelView-like exploration interface for non-standard relation algebras.

The Use of Haskell

Haskell [HPJW$^+$92] is a purely functional programming language and is currently widely accepted in research and university teaching. The fact that Haskell is a referentially transparent programming language makes it particularly suitable for dealing with mathematical structures and treating them as immutable entities.

This *safety* together with the abstraction support provided by higher-order functions make Haskell an ideal language for the definition and exploration of new structures. The current report strives to provide a toolkit that lends itself easily to this task. Every special investigation will of course need its own extensions; even with only superficial knowledge of Haskell it should be possible to build customised tools.

For more information about Haskell see the Haskell WWW site at http://www.haskell.org/ (there you also find links to implementations), or the Journal of Functional Programming.

Acknowledgements

We gratefully acknowledge the input from our friends and colleagues from the RelMiCS community that greatly helped shaping this report, in particular Michael Winter, who pioneered with precursors of the current toolkit.

Chapter 1

Relation Algebra Definition and Exploration

Since the tool-set described in in this report is geared towards working with non-standard models of relation algebras, we decided to also support weaker mathematical structures, since something that is almost-but-not-quite a relation algebra might still prove useful in the search for relation algebras with unusual properties.

The obvious candidates for these structures may be taken from the hierarchy of *allegories* defined by Freyd and Scedrov [FS90], including categories as the basis.

1.1 From Categories to Relation Algebras

In this section we review the necessary definitions and a few of their properties; the notation we use is that agreed upon for the book [BKS97].

Alongside, we introduce Haskell identifiers that shall serve to access the different components of these mathematical structures in our Haskell-based exploration system. As most sections of this report, the file containing the present section is therefore at the same time a *literate* Haskell module.

This present module offers a uniform interface to categories of all levels, but at the cost of employing a non-standard extension to Haskell 98, namely *multi-parameter type classes* with *functional dependencies* between parameters [Jon00] — these are currently supported by the Haskell interpreter Hugs[1].

For Haskell, this module begins with the following heading:

```
module RelAlgClasses where
```

The absence of an explicit export list before the keyword `where` implies that everything defined in this module is also exported.

1.1.1 Categories

We recall the definition of a category, the construct that we have chosen to model heterogeneity of relation algebras.

Definition 1.1.1 A *category* $\mathbf{C}$ is a tuple $(Obj_{\mathbf{C}}, Mor_{\mathbf{C}}, \_ : \_ \rightarrow \_, \mathbb{I}, ;)$ where

- $Obj_{\mathbf{C}}$ is a collection of *objects*.

[1]To load this module, Hugs needs to be started with the command line option "`-98`" which enables Hugs-specific extensions; this option cannot be changed while the interpreter is running.

- $Mor_{\mathbf{C}}$ is a collection of *arrows* or *morphisms.*
- "$\_ : \_ \to \_$" is ternary relation relating every morphism f univalently with two objects $\mathcal{A}$ and $\mathcal{B}$, written $f : \mathcal{A} \to \mathcal{B}$, where $\mathcal{A}$ is called the *source* of f, and $\mathcal{B}$ the *target* of f.

  The collection of all morphisms f with $f : \mathcal{A} \to \mathcal{B}$ is denoted as $Hom_{\mathbf{C}}[\mathcal{A}, \mathcal{B}]$ and also called a *homset.*
- "$;$" is the binary *composition* operator, and composition of two morphisms $f : \mathcal{A} \to \mathcal{B}$ and $g : \mathcal{B}' \to \mathcal{C}$ is defined iff $\mathcal{B} = \mathcal{B}'$, and then $(f;g) : \mathcal{A} \to \mathcal{C}$; composition is associative.
- $\mathbb{I}$ associates with every object $\mathcal{A}$ a morphism $\mathbb{I}_{\mathcal{A}}$ which is both a right and left unit for composition. □

Composition operators like "$;$" will bind with a higher priority than all other binary operators.

For being able to manipulate categories as data in Haskell programs, we define a multi-parameter class `Category` with three parameters: The type variable `cat` stands for the type of categories-as-data, and we do not parameterise this type for the time being. Next, `obj` is the type of objects, but this type need not exclusively comprise objects of the categories in question, so we add a member predicate `isObj` that checks whether some item of the object type is an object of the category in question. Last, `mor` is the type of morphisms, and the test whether some item of the morphism type is in fact a morphism is specialised to directly check membership in the homset spanned by two objects, i.e., membership in the relation $\_ : \_ \to \_$. This is more useful than a global morphism test (which could be defined using the `source` and `target` functions), and the obligation to provide the additional source and target arguments seems not to be molesting in our experience.

Since we do exhaustive exploration rather than symbolic proofs, we want to treat only finite categories, i.e., categories where both $Obj_{\mathbf{C}}$ and $Mor_{\mathbf{C}}$ are finite sets. Therefore we demand an enumeration `objects` of the object set and, for every two objects `s` and `t`, an enumeration `homset s t` of the corresponding homset.

For theoretical purposes, the important restriction here is only that homsets should be finite, so we define:

Definition 1.1.2 When all homsets of a category are restricted to be sets, the category is called *locally small.* A locally small category is called *locally finite* if every homset is a finite set. □

The remaining two class members are in direct correspondence to items of the mathematical definition; given the enumerations of objects and homsets it is of course possible to derive the identities from the other information, but we shall generally postpone such decisions to the implementation. The interface is much easier to use if certain derived components are included directly in the interface, and this way, also the implementation has more freedom to use more efficient definitions.

```
class Category cat obj mor | cat -> obj, cat -> mor where
  isObj   :: cat -> obj -> Bool
  isMor   :: cat -> obj -> obj -> mor -> Bool
  objects :: cat -> [obj]
  homset  :: cat -> obj -> obj -> [mor]
  source  :: cat -> mor -> obj
  target  :: cat -> mor -> obj
  idmor   :: cat -> obj -> mor
  comp    :: cat -> mor -> mor -> mor
```

The functional dependencies "`cat -> obj, cat -> mor`" correspond to the fact that every category $\mathbf{C}$ brings with it the type of its objects and morphisms. These functional dependencies are necessary because the parameter variable `obj` does not occur in the type of `comp`, and `mor` does not occur in the type of `objects`.

1.1.2 Allegories

The simplest abstraction of the behaviour of relations among those presented in [FS90] only reflects transposition (converse) and intersection (meet) (and therewith also inclusion) of relations on top of the category structure:

Definition 1.1.3 An *allegory* is a tuple $\mathbf{C} = (Obj_{\mathbf{C}}, Mor_{\mathbf{C}}, \_ : \_ \leftrightarrow \_, \mathbb{I}, ;, \breve{}\,, \sqcap)$ where:

i) The tuple $(Obj_{\mathbf{C}}, Mor_{\mathbf{C}}, \_ : \_ \leftrightarrow \_, \mathbb{I}, ;)$ is a category, the so-called *underlying category* of $\mathbf{C}$.[2] The morphisms are usually called *relations*.

ii) Every homset $Hom_{\mathbf{C}}[\mathcal{A}, \mathcal{B}]$ carries the structure of a lower semi-lattice with $\sqcap_{\mathcal{A},\mathcal{B}}$ for *meet*, and inclusion ordering $\sqsubseteq_{\mathcal{A},\mathcal{B}}$, all usually written without indices.

iii) $\breve{}$ is the total unary operation of *conversion* of morphisms, where for $R : \mathcal{A} \leftrightarrow \mathcal{B}$ we have $R\breve{} : \mathcal{B} \leftrightarrow \mathcal{A}$, and the following properties hold:
   (a) $(R\breve{})\breve{} = R$,
   (b) $(Q;R)\breve{} = R\breve{};Q\breve{}$,
   (c) $(Q \sqcap Q')\breve{} = Q\breve{} \sqcap Q'\breve{}$.

iv) For all $Q : \mathcal{A} \leftrightarrow \mathcal{B}$ and $R, R' : \mathcal{B} \leftrightarrow \mathcal{C}$, *meet-subdistributivity* holds:

$$Q;(R \sqcap R') \sqsubseteq Q;R \sqcap Q;R' \ .$$

v) For all $Q : \mathcal{A} \leftrightarrow \mathcal{B}$, $R : \mathcal{B} \leftrightarrow \mathcal{C}$, and $S : \mathcal{A} \leftrightarrow \mathcal{C}$, the *modal rule* holds:

$$Q;R \sqcap S \sqsubseteq (Q \sqcap S;R\breve{});R \ . \qquad \square$$

[2] $Mor_{\mathbf{C}}$ may be a class in [FS90], meaning that there, allegories are not restricted to be locally small. The price of this generality, however, is that join, meet, etc. need to be characterised at a more elementary level, while we can introduce these as lattice operators. Since we mostly have finite categories in mind, anyway, we may sacrifice that generality for the sake of brevity and readability.

We define the type class `Allegory` as a sub-class of `Category`, adding the converse and meet operators, and the inclusion relation between morphisms:

```
class Category all obj mor => Allegory all obj mor | all -> obj, all -> mor where
  converse :: all -> mor -> mor
  meet     :: all -> mor -> mor -> mor
  incl     :: all -> mor -> mor -> Bool
```

The following basic properties are easily deduced from the definition of allegories:

- Conversion is an isotone and involutive contravariant functor: In addition to the properties from the definition, this comprises also $\mathbb{I}_A^{\smile} = \mathbb{I}_A$ and $Q \sqsubseteq Q' \Leftrightarrow Q^{\smile} \sqsubseteq Q'^{\smile}$.
- Composition is monotonic: If $Q \sqsubseteq Q'$ and $R \sqsubseteq R'$, then $Q{;}R \sqsubseteq Q'{;}R'$.

From the modal rule listed among the allegory axioms, we may — using properties of conversion — obtain the other modal rule

$$Q{;}R \sqcap S \sqsubseteq Q{;}(R \sqcap Q^{\smile}{;}S) \enspace ,$$

which is used by Olivier and Serrato for their axiomatisation of Dedekind categories [OS80, OS95] (see also the next section) and there called Dedekind formula — by Jacques Riguet, however, this name had much earlier been attached to the formula proved in the next proposition [Rig48]. Paul Lorenzen called it *Bund-Axiom* [Lor54].

Proposition 1.1.4 Both *modal rules*

$$\begin{array}{rcll} Q{;}R \sqcap S & \sqsubseteq & Q{;}(R \sqcap Q^{\smile}{;}S) & (m1) \\ Q{;}R \sqcap S & \sqsubseteq & (Q \sqcap S{;}R^{\smile}){;}R & (m2) \end{array}$$

together are equivalent to the *Dedekind rule*

$$Q{;}R \sqcap S \sqsubseteq (Q \sqcap S{;}R^{\smile}){;}(R \sqcap Q^{\smile}{;}S) \enspace .$$

Proof: The modal rules follow immediately from the Dedekind rule:

$$Q{;}R \sqcap S \sqsubseteq (Q \sqcap S{;}R^{\smile}){;}(R \sqcap Q^{\smile}{;}S) \sqsubseteq \left\{ \begin{array}{l} (Q \sqcap S{;}R^{\smile}){;}R \\ Q{;}(R \sqcap Q^{\smile}{;}S) \end{array} \right.$$

Conversely, assume that the modal rules hold. Then we have

$$\begin{array}{rcll} Q{;}R \sqcap S & \sqsubseteq & Q{;}(R \sqcap Q^{\smile}{;}S) \sqcap S & \text{(m1)} \\ & \sqsubseteq & (Q \sqcap S{;}(R \sqcap Q^{\smile}{;}S)^{\smile}){;}(R \sqcap Q^{\smile}{;}S) & \text{(m2)} \\ & \sqsubseteq & (Q \sqcap S{;}R^{\smile}){;}(R \sqcap Q^{\smile}{;}S) \enspace . & \forall U, V : U \sqcap V \sqsubseteq V \end{array}$$

□

1.1.3 Distributive Allegories

To the structure presented so far, we now add the possibility of finding joins and a zero together with distributivity of composition over joins.

Definition 1.1.5 A *distributive allegory* is a tuple
$\mathbf{C} = (Obj_{\mathbf{C}}, Mor_{\mathbf{C}}, \_ : \_ \leftrightarrow \_, \mathbb{I}, ;, \breve{}, \sqcap, \sqcup, \perp\!\!\!\perp)$ where the following hold:

i) The tuple $(Obj_{\mathbf{C}}, Mor_{\mathbf{C}}, \_ : \_ \leftrightarrow \_, \mathbb{I}, ;, \breve{}, \sqcap)$ is an allegory, the so-called *underlying allegory* of $\mathbf{C}$.

ii) Every homset $Hom_{\mathcal{C}}[\mathcal{A}, \mathcal{B}]$ carries the structure of a distributive lattice with $\sqcup_{\mathcal{A},\mathcal{B}}$ for *join*, and zero element $\perp\!\!\!\perp_{\mathcal{A},\mathcal{B}}$.

iii) For all objects $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$ and all morphisms $Q : \mathcal{A} \leftrightarrow \mathcal{B}$, the *zero law* holds:
$$Q;\perp\!\!\!\perp_{\mathcal{B},\mathcal{C}} = \perp\!\!\!\perp_{\mathcal{A},\mathcal{C}} \quad .$$

iv) For all $Q : \mathcal{A} \leftrightarrow \mathcal{B}$ and $R, R' : \mathcal{B} \leftrightarrow \mathcal{C}$, *join-distributivity* holds:
$$Q;(R \sqcup R') = Q;R \sqcup Q;R' \quad . \qquad \square$$

We mention a few easily derivable facts.

Proposition 1.1.6 Let $Q, Q' : \mathcal{A} \leftrightarrow \mathcal{B}$ and $R : \mathcal{B} \leftrightarrow \mathcal{C}$ be morphisms in a distributive allegory. Then:

i) $\perp\!\!\!\perp\breve{}_{A,B} = \perp\!\!\!\perp_{B,A}$.

ii) $\perp\!\!\!\perp_{A,B};R = \perp\!\!\!\perp_{A,C}$.

iii) $(Q \sqcup Q')\breve{} = Q\breve{} \sqcup Q'\breve{}$. $\square$

For our Haskell module, the new class `DistribAllegory` only needs to add two components:

```
class        Allegory all obj mor =>
      DistribAllegory all obj mor | all -> obj, all -> mor where
  join   :: all -> mor -> mor -> mor
  bottom :: all -> obj -> obj -> mor
```

Distributive allegories with only finite homsets are locally complete, according to the definition of [FS90, 2.22]:

Definition 1.1.7 A distributive allegory is *locally complete* if every homset is a complete lattice, and if composition and finite intersection distribute over arbitrary unions: that is, given $R : \mathcal{A} \leftrightarrow \mathcal{B}$ and $\{S_i : \mathcal{B} \leftrightarrow \mathcal{C}\}_{i \in I}$ one has $R;(\bigsqcup_{i \in I} S_i) = \bigsqcup_{i \in I}(R;S_i)$. For empty I we understand this to mean $R;\perp\!\!\!\perp = \perp\!\!\!\perp$. $\square$

1.1.4 Division Allegories

Demanding properties usually attributed to a division operation characterises division allegories among distributive allegories.

Definition 1.1.8 [FS90] A *division allegory* is a distributive allegory where for arbitrary relations $S : \mathcal{A} \leftrightarrow \mathcal{C}$ and $R : \mathcal{B} \leftrightarrow \mathcal{C}$, the *left residual* S/R exists, defined by

$$Q;R \sqsubseteq S \iff Q \sqsubseteq S/R \qquad \text{for all } Q : \mathcal{A} \leftrightarrow \mathcal{B} \ . \qquad \square$$

On top of the left residual we may continue to define:

Definition 1.1.9 In a division allegory, the *right residual* may be defined via the left residual:

$$Q\backslash S := (S\breve{}/Q\breve{})\breve{}$$

and fulfils a corresponding specification:

$$Q;R \sqsubseteq S \iff R \sqsubseteq Q\backslash S \qquad \text{for all } R : \mathcal{B} \leftrightarrow \mathcal{C}$$

The *symmetric quotient* is defined as the intersection of two residuals: For $P : \mathcal{A} \leftrightarrow \mathcal{B}$ and $Q : \mathcal{A} \leftrightarrow \mathcal{C}$ we have $\mathrm{syq}(P, Q) : \mathcal{B} \leftrightarrow \mathcal{C}$ with

$$\mathrm{syq}(R, S) = R\backslash S \sqcap R\breve{}/S\breve{} \ . \qquad \square$$

This symmetric quotient has originally been defined in the context of heterogeneous relation algebras [BSZ86, BSZ89] and is — modulo conversion of the arguments — exactly the *symmetric division* as introduced by Freyd and Scedrov for division allegories [FS90, 2.35].

For concrete relations R and S, the symmetric quotient relates elements r from the range of R with elements s from the range of S exactly if the inverse image of r under R is the same as the inverse image of s under S, or, in the language of predicate logic:

$$(r, s) \in \mathrm{syq}(R, S) \qquad \iff \qquad \forall x : (x, r) \in R \leftrightarrow (x, s) \in S$$

(Riguet had introduced the unary operation of "noyeau" in the homogeneous setting, which can now be seen as defined by $\mathsf{noy}(R) = \mathrm{syq}(R, R)$, in [Rig48].)

```
class  DistribAllegory all obj mor =>
      DivisionAllegory all obj mor | all -> obj, all -> mor where
  rres :: all -> mor -> mor -> mor
  lres :: all -> mor -> mor -> mor
  syq  :: all -> mor -> mor -> mor
```

The conditions of meet-subdistributivity, join-distributivity and zero law listed for distributive allegories are not required in the axiomatisation of division allegories, since here they can be deduced using the residuals.

On the other hand, residuals always exist in a locally complete distributive allegory, so every locally finite distributive allegory is a division allegory.

1.1.5 Dedekind Categories

Independent of Freyd and Scedrov, Olivier and Serrato defined a kind of relation categories in [OS80] which differs from division allegories precisely by being what is called “locally complete” in [FS90, 2.22]:

Definition 1.1.10 [OS80] A *Dedekind category* is a division allegory $\mathcal{C}$ where every homset $Hom_{\mathcal{C}}[A,B]$ is a *complete* lattice with greatest element $\mathbb{T}_{A,B}$, called *universal relation.* □

```
class DivisionAllegory ded obj mor =>
      DedCat           ded obj mor | ded -> obj, ded -> mor where
  top  :: ded -> obj -> obj -> mor
```

In contrast to [FS90, 2.22], the infinite variants of meet-subdistributivity and join-distributivity, which form part of the definition of local completeness, need not be listed here, since they follow from the complete lattice structure via the presence of residuals. On the other hand, the full definition of local completeness implies the existence of residuals [FS90, 2.315], such that a Dedekind category is just a locally complete distributive allegory.

We still separate the Haskell definitions of distributive allegories, division allegories and Dedekind categories since these Haskell definitions themselves are equally adequate to deal with infinite structures, and may also prove useful for that purpose. It is only our tests that rely on finiteness.

1.1.6 Relation Algebras

If all morphisms of a Dedekind category have complements, the Dedekind category is equivalent to a Schröder category:

Definition 1.1.11 A *Schröder category* [OS80, Jón88] is a Dedekind category where every homset is a Boolean lattice. □

The complement of a relation R is written $\overline{R}$.

It is well-known that in a distributive allegory with Boolean lattices as homsets, the Dedekind rule is equivalent to the *Schröder equivalences*:

$$Q{;}R \sqsubseteq S \qquad \Longleftrightarrow \qquad Q^{\smile}{;}\overline{S} \sqsubseteq \overline{R} \qquad \Longleftrightarrow \qquad \overline{S}{;}R^{\smile} \sqsubseteq \overline{Q}$$

for all relations $Q : \mathcal{A} \leftrightarrow \mathcal{B}$, $R : \mathcal{B} \leftrightarrow \mathcal{C}$ and $S : \mathcal{A} \leftrightarrow \mathcal{C}$. For the first direction, it is sufficient to show that with the Dedekind rule, $Q{;}R \sqsubseteq S$ implies $Q^{\smile}{;}\overline{S} \sqsubseteq \overline{R}$: assume $Q{;}R \sqsubseteq S$, then that is equivalent to $Q{;}R \sqcap \overline{S} = \mathbb{L}$, and we have

$$Q^{\smile}{;}\overline{S} \sqcap R \sqsubseteq Q^{\smile}{;}(\overline{S} \sqcap Q{;}R) = \mathbb{L} \quad .$$

Conversely, assume that the Schröder equivalences hold. Then [SS85b] shows:

$$\begin{array}{lll}
 & Q;R & \\
= & ((Q \sqcap S;R^{\smile}) \sqcup (Q \sqcap \overline{S;R^{\smile}}));((R \sqcap Q^{\smile};S) \sqcup (R \sqcap \overline{Q^{\smile};S})) & \text{Boolean lattice} \\
= & (Q \sqcap S;R^{\smile});(R \sqcap Q^{\smile};S) \sqcup (Q \sqcap S;R^{\smile});(R \sqcap \overline{Q^{\smile};S}) & \\
 & \sqcup\ (Q \sqcap \overline{S;R^{\smile}});(R \sqcap Q^{\smile};S) \sqcup (Q \sqcap \overline{S;R^{\smile}});(R \sqcap \overline{Q^{\smile};S}) & \text{join-distributivity} \\
\sqsubseteq & (Q \sqcap S;R^{\smile});(R \sqcap Q^{\smile};S) \sqcup Q;\overline{Q^{\smile};S} \sqcup \overline{S;R^{\smile}};R & \forall U, V\ : U \sqcap V \sqsubseteq U \\
\sqsubseteq & (Q \sqcap S;R^{\smile});(R \sqcap Q^{\smile};S) \sqcup \overline{S} & \text{Schröder}
\end{array}$$

yielding the Dedekind rule $Q;R \sqcap S \sqsubseteq (Q \sqcap S;R^{\smile});(R \sqcap Q^{\smile};S)$ via Boolean lattice properties.

Furthermore, the Schröder equivalences allow us to calculate:

$$Q;R \sqsubseteq S \qquad \Longleftrightarrow \qquad \overline{S};R^{\smile} \sqsubseteq \overline{Q} \qquad \Longleftrightarrow \qquad Q \sqsubseteq \overline{\overline{S};R^{\smile}}$$

Therefore, we have $S/R = \overline{\overline{S};R^{\smile}}$, so that in Schröder categories the residual is defined *a priori* and need not be listed in the axiomatisation.

The concept of Schröder categories can be considered as a slightly relaxed variant of the following, older, concept of heterogeneous relation algebras:

Definition 1.1.12 A *heterogeneous relation algebra* [Sch77, Sch81a, SS89, SS93] is a Schröder category where every homset is an *atomic* and complete Boolean lattice. □

In many contexts, non-triviality of the Boolean lattices is also demanded, namely $\mathbb{T}_{\mathcal{A},\mathcal{B}} \neq \perp\!\!\!\perp_{\mathcal{A},\mathcal{B}}$ for all objects $\mathcal{A}$ and $\mathcal{B}$, and also the following rule:

Definition 1.1.13 The *Tarski rule* holds in a heterogeneous relation algebra iff

$$R \neq \perp\!\!\!\perp_{\mathcal{A},\mathcal{B}} \quad \Longleftrightarrow \quad \mathbb{T}_{\mathcal{C},\mathcal{A}};R;\mathbb{T}_{\mathcal{B},\mathcal{D}} = \mathbb{T}_{\mathcal{C},\mathcal{D}}$$

holds for all $\mathcal{A}, \mathcal{B}, \mathcal{C}, \mathcal{D} : \mathsf{Obj}_{\mathcal{R}}$ and $R : \mathcal{A} \leftrightarrow \mathcal{B}$. □

Both of these constraints, however, are inappropriate for our search for computationally relevant non-standard relation algebras, so they are not included in the definition here.

Obviously, a Schröder category with finite homsets is always a heterogeneous relation algebra, so we directly introduce an interface for the latter:

```
class DedCat ra obj mor => RelAlg ra obj mor | ra -> obj, ra -> mor where
  compl :: ra -> mor -> mor
```

1.2 Data Structures and Tests

Although the multi-parameter-class interface presented in the last section might look quite attractive at first sight, it has several drawbacks. Most obviously, the use of a non-standard extension to Haskell brings about portability problems. Also, since there is not yet a universally accepted design for multi-parameter classes, their use is always prone to future changes of supporting implementations.

The natural solution would be an ML-style module system. Since this is not available in Haskell, we resort to a translation of module types into record data types, where types contained in the module become type parameters of the record type constructor.

This approach makes the translation of the classes of the last section into explicit *dictionary records* straightforward. We use prefixes to separate the name spaces, and to ease a class-like use, we explicitly import superclass members into subclasses by straightforward selector composition.

We use abbreviated type names in order to avoid name conflicts with the class names of the previous section, since, in Sect. 1.4, we are going to enable access to the constructions presented here via those class interfaces. In implementations that support multi-parameter type classes with functional dependencies (such as Hugs), we can therefore seamlessly integrate the class view of the last section and the explicit dictionary view of this section.

1.2.1 Preliminaries

This is the central module of our relation algebra library, and there is nothing to hide here. We do, however, import a few utilities from Haskell's standard libraries, on from our own prelude extensions `ExtPrel` listed in Sect. A.3:

```
module RelAlg where

import qualified IO(hFlush, stdout)
import Maybe(listToMaybe)
import ExtPrel(listEqAsSet)
```

1.2.2 Testing

We shall define numerous tests that allow to check whether the structures we introduce are well-defined, or whether certain laws hold or not.

In either case, a negative result should indicate in which way the test failed, so we define a uniform test result structure that can hold all information for a single failure case in the context of a single category or relation algebra:

```
type Instance obj mor = (String,[obj],[mor])
```

The semantics of such an `Instance` does of course heavily depend on its production site. But we find that this is detailed and simple enough both for the test programmer and for the test user.

We display `Instances` with every component on a line of its own:

```
showsInstance :: (Show obj, Show mor) => Instance obj mor -> ShowS
showsInstance (s,os,ms) r = foldr (\s' r' -> s' ++ '\n' : r') r
   (s : case os of [] -> []
                   [o] -> [" Object: " ++ show o]
                   _ -> " Objects:"   : map (indent . show) os
     ++ case ms of [] -> []
                   [m] -> [" Morphism: " ++ show m]
                   _ -> " Morphisms:" : map (indent . show) ms)
  where indent s' = "  " ++ s'

showInstance :: (Show obj, Show mor) => Instance obj mor -> String
showInstance i = showsInstance i ""
```

A simplistic approach would let individual test cases produce results of type `[Instance]` and then concatenate these to the complete test result. Since concatenation may incur quadratic running time costs, we use the standard technique to replace concatenation with function composition and let individual test cases return results of the following type (in analogy to the prelude type `ShowS = String -> String`):

```
type TestResult obj mor = [Instance obj mor] -> [Instance obj mor]
```

As in the case of `ShowS`, function composition now acts as a low-cost binary concatenation operator on expressions of type `TestResult`.

Typically, we shall generate `TestResults` via the following function:

```
test :: Bool -> [obj] -> [mor] -> String -> TestResult obj mor
test b os ms s = \ is -> if b then is else (s,os,ms) : is
```

Sometimes, however, presence of a result is an indication that certain other tests need not be performed; for these circumstances we provide a variant operating on lazy `TestResult` lists:

```
testX :: Bool -> [obj] -> [mor] -> String ->
         [TestResult obj mor] -> [TestResult obj mor]
testX b os ms s = \ crs -> if b then crs else [((s,os,ms):)]
```

As in the case of `ShowS`, functions of type `TestResult` never inspect their argument, but return it with maybe some additional `Instance`s consed onto its beginning.

For testing any individual property, usually a whole list of `TestResults` is produced, and we concatenate them with the following instance of `foldr`:

```
ffold :: [a -> a] -> a -> a
ffold l r = foldr id r l
```

In our tests, we then use this at the type

```
ffold :: [TestResult obj mor] -> TestResult obj mor.
```

For tests that check structures like categories or relation algebras for consistency or for occurrence of certain special configurations, we then may use the following type, where `s` is a binary type constructor:

```
type Test s obj mor = s obj mor -> TestResult obj mor
```

The most frequent use of tests will be to *perform* them interactively for inspecting the results:

```
perform :: (Show obj, Show mor) => Test c obj mor -> c obj mor -> IO ()
perform t c = printTestResults (t c)
```

Since the output for every instance may be quite verbose, we currently only output the first three test result `Instance`s:

```
printTestResults :: (Show obj, Show mor) => TestResult obj mor -> IO ()
printTestResults t = case map showsInstance $ t [] of
                       [] -> putStrLn "No results."
                       l -> putStr $ ffold (take 3 l) ""
```

For situations where all results are needed, we also provide:

```
performAll :: (Show obj, Show mor) => Test c obj mor -> c obj mor -> IO ()
performAll t c = printAllTestResults (t c)

printAllTestResults :: (Show obj, Show mor) => TestResult obj mor -> IO ()
printAllTestResults t = do
  putStrLn "=== Test Start ==="
  mapM_ (putStrFlush . showInstance) (t [])
  putStrLn "=== Test End   ==="

putStrFlush s = putStr s >> IO.hFlush IO.stdout
```

Sometimes, however, we are only interested whether there are any results or not:

```
noResults :: Test c obj mor -> c obj mor -> Bool
noResults t c = null (t c [])
```

1.2.3 Categories

We now turn to the data structures representing categories etcetera. We define them as record data types, using as field labels (i.e. also as selector functions) the corresponding method names prefixed with a lower-case variant of the type name (which we also use as the constructor name):

```
data Cat obj mor = Cat
  {cat_isObj   :: obj -> Bool
  ,cat_isMor   :: obj -> obj -> mor -> Bool
  ,cat_objects :: [obj]
  ,cat_homset  :: obj -> obj -> [mor]
  ,cat_source  :: mor -> obj
  ,cat_target  :: mor -> obj
  ,cat_idmor   :: obj -> mor
  ,cat_comp    :: mor -> mor -> mor
  }
```

We organise the consistency test for categories into four groups:

i) One object: Consistency of object list and of identity as a morphism

ii) Two objects, one morphism: Consistency of morphism list, identity properties

iii) Three objects, two morphisms: Well-definedness of composition

iv) Four objects, three morphisms: Associativity of composition

We generate the result lists via do expressions in the list monad; with respect to list comprehension this has the advantage that local variables are introduced *before* they are used. Since return in the list monad is just the singleton function, we usually directly write singletons instead of return since this saves space and serves as an additional reminder that the do expressions are in the list monad.

Keeping the tests in separate do expressions has the advantage of better readability, and also the advantage that different failures of one property are grouped closer together. However it incurs a slight runtime cost. Later we will usually join the tests of different complexity into nested do expressions. Then, failures will be grouped essentially according to the objects and morphisms involved in them.

All the tests included in this report are decision procedures for finite categories. Although it is perfectly possible to use diagonalisation to obtain semi-decision procedures for countable categories, the overhead would incur significant running-time and readability costs for the finite case, which is the case we are interested in.

```
cat_TEST :: (Eq obj, Eq mor) => Test Cat obj mor
cat_TEST c =
  let isObj = cat_isObj c
      isMor = cat_isMor c
      objs = cat_objects c
      homset = cat_homset c
      source = cat_source c
      target = cat_target c
      idmor = cat_idmor c
      (^) = cat_comp c
  in ffold (let a1 = "identity "
              a2 = a1 ++ "has inconsistent " in
          do o <- objs
             let i = idmor o
             [test (isObj  o     ) [o] [] "object list contains non-object" .
              test (source i == o) [o] [i] (a2 ++ "source") .
              test (target i == o) [o] [i] (a2 ++ "target") .
              test (isMor  o o  i) [o] [i] (a1 ++ "is non-morphism")]
         ) .
   ffold (let a1 = "homset contains "
              a2 = a1 ++ "morphism with inconsistent " in
          do s <- objs
             let sId = idmor s
             t <- objs
             let os = [s,t]
             let tId = idmor t
             m <- homset s t
             [test (source m == s) os [m] (a2 ++ "source") .
              test (target m == t) os [m] (a2 ++ "target") .
              test (isMor s t m  ) os [m] (a1 ++ "non-morphism") .
              test (sId ^ m == m) os [sId,m] "left-identity violated" .
              test (m ^ tId == m) os [m,tId] "right-identity violated"]
         ) .
   ffold (let a1 = "composition yields "
              a2 = a1 ++ "morphism with inconsistent " in
          do o1 <- objs
             o2 <- objs
             f <- homset o1 o2
             o3 <- objs
             let os = [o1,o2,o3]
             g <- homset o2 o3
             let m = f ^ g
             let ms = [f,g,m]
             [test (source m == o1) os ms (a2 ++ "source") .
              test (target m == o3) os ms (a2 ++ "target") .
              test (isMor o1 o3 m ) os ms (a1 ++ "non-morphism")]
         ) .
   ffold (do o1 <- objs
             o2 <- objs
             f <- homset o1 o2
             o3 <- objs
             g <- homset o2 o3
             let fg = f ^ g
             o4 <- objs
             let os = [o1,o2,o3,o4]
             h <- homset o3 o4
             let gh = g ^ h
             let k1 = f ^ gh
             let k2 = fg ^ h
```

```
                [test (k1 == k2) os [f,g,h,fg,gh,k1,k2]
                     "non-associative composition"    ]
            )
```

1.2.4 Functors

Functors are category homomorphisms and therefore an important tool for establishing relations between different categories. Unfortunately, the prelude defines "`Functor`" as class name for endofunctors in the category of Haskell types and Haskell functions — we resolve the name clash with the prelude class `Functor` by using the abbreviation `Fun`.

Furthermore, for the time being we want to work with the `Test` datatype from above, and we want the objects and morphisms of the source category to appear in the `TestResults` — this determines the reversed order of the type arguments to the `Fun` type constructor:

```
data Fun obj2 mor2 obj1 mor1 = Fun
  {fun_obj :: obj1 -> obj2
  ,fun_mor :: mor1 -> mor2
  }
```

Since we align the direction of functor composition with the direction of our categorical composition, the twisted type of functors "recovers" the usual twisted type of composition:

```
funcomp :: Fun obj2 mor2 obj1 mor1 ->
           Fun obj3 mor3 obj2 mor2 ->
           Fun obj3 mor3 obj1 mor1
Fun fo1 fm1 `funcomp` Fun fo2 fm2  =  Fun (fo1 $$$ fo2) (fm1 $$$ fm2)

($$$) :: (a -> b) -> (b -> c) -> (a -> c)
f $$$ g = \ x -> g (f x)
```

Testing whether a functor data structure does indeed represent a functor is divided into three steps:

i) One object: Testing well-formedness of the object mapping, and preservation of identities

ii) Two objects, one morphism: Testing well-formedness of the morphism mapping

iii) Three objects, two morphisms: Testing preservation of composition.

```
functor_TEST :: Eq mor2 => Cat obj1 mor1 -> Cat obj2 mor2 ->
                           Test (Fun obj2 mor2) obj1 mor1
functor_TEST c1 c2 fun =
  let objects1 = cat_objects c1
      homset1 = cat_homset c1
      idmor1 = cat_idmor c1
      (^) = cat_comp c1

      isObj2 = cat_isObj c2
      isMor2 = cat_isMor c2
      idmor2 = cat_idmor c2
      (^^) = cat_comp c2

      fo = fun_obj fun
      fm = fun_mor fun
  in ffold (do
   s1 <- objects1
   let s2 = fo s1
   let is1 = idmor1 s1
   let is2 = idmor2 s2
   testX (isObj2 s2) [s1] [] "functor yields non-object" $
    [test (fm is1 == is2) [s1] [is1] "functor does not preserve identity"]
  ) . ffold (do
   s1 <- objects1
   let s2 = fo s1
   t1 <- objects1
   let t2 = fo t1
   f1 <- homset1 s1 t1
   [test (isMor2 s2 t2 (fm f1)) [s1, t1] [f1] "functor yields non-morphism"]
  ) . ffold (do
   s1 <- objects1
   t1 <- objects1
   f1 <- homset1 s1 t1
   let f2 = fm f1
   u1 <- objects1
   g1 <- homset1 t1 u1
   let g2 = fm g1
   let h1 = f1 ^ g1
   [test (fm h1 == (f2 ^^ g2)) [s1,t1,u1] [f1,g1]
         "functor does not preserve composition" ]
  )
```

We also implement a straightforward test for checking whether some other functor `f2` is right-inverse with respect to ‘`funcomp`‘ to the test argument `f1`:

```
functor_rightinv_test :: (Eq mor1, Eq obj1) =>
                         Cat obj1 mor1 -> Cat obj2 mor2 ->
                         Fun obj1 mor1 obj2 mor2 ->
                         Test (Fun obj2 mor2) obj1 mor1
functor_rightinv_test c1 c2 f2 f1 =
  let objects1 = cat_objects c1
      homset1 = cat_homset c1
      fo1 = fun_obj f1
      fo2 = fun_obj f2
      fm1 = fun_mor f1
      fm2 = fun_mor f2
  in ffold (do
   o1 <- objects1
   let o2 = fo1 o1
   let o1a = fo2 o2
   [test (o1 == o1a) [o1,o1a] [] "not right-inverse on objects"]
  ) . ffold (do
   o1 <- objects1
   o2 <- objects1
   f1 <- homset1 o1 o2
   let f2 = fm1 f1
   let f1a = fm2 f2
   [test (f1 == f1a) [o1,o2] [f1,f1a] "not right-inverse on morphisms"]
  )
```

1.2.5 Allegories

We already mentioned that records like those of the `Cat obj mor` datatype correspond to method dictionaries. We now proceed to define the first subclass of categories, and we include the `Cat` dictionary as first entry in the subclass dictionary, or, mathematically, explicitly include the base category as such:

```
data All obj mor = All
  {all_cat  :: Cat obj mor
  ,all_converse :: mor -> mor
  ,all_meet     :: mor -> mor -> mor
  ,all_incl     :: mor -> mor -> Bool
  }
```

For transparent access to all parts of the mathematical structure (corresponding to the flat tuples in the definitions) we transfer the superclass methods into the subclass via composition with the superclass dictionary selector:

```
all_isObj   = cat_isObj   . all_cat      -- :: obj -> Bool
all_isMor   = cat_isMor   . all_cat      -- :: obj -> obj -> mor -> Bool
all_objects = cat_objects . all_cat      -- :: [obj]
all_homset  = cat_homset  . all_cat      -- :: obj -> obj -> [mor]
all_source  = cat_source  . all_cat      -- :: mor -> obj
all_target  = cat_target  . all_cat      -- :: mor -> obj
all_idmor   = cat_idmor   . all_cat      -- :: obj -> mor
all_comp    = cat_comp    . all_cat      -- :: mor -> mor -> mor
```

The consistency tests for allegories are organised into two large groups:

i) Two objects:

   (a) One morphism: Consistency of converse and idempotency of meet
   (b) Two morphisms: Consistency and commutativity of meet, monotony of converse and consistency with meet
   (c) Three morphisms: Associativity and sub-distributivity of meet

ii) Three objects:

   (a) Two morphisms: Preservation of composition by converse
   (b) Three morphisms: modal rule

The tests that are commented out are for properties that are implied by the other tests.

```
all_TEST :: (Eq obj, Eq mor) => Test All obj mor
all_TEST c =
  let (^) = all_comp c
      conv = all_converse c
      (<<==) = all_incl c
      (&&&) = all_meet c
      homset = all_homset c
      objs = all_objects c
      idmor = all_idmor c
      convNPres = "converse does not preserve "
  in -- ffold (do s <- objs
     --          let i = idmor s
     --          let i' = conv i
     --          [test (i == i') [s] [i,i'] (convNPres ++ "identity")]
     --       ) .
     ffold (let c1 = "converse yields "
                c2 = c1 ++ "morphism with inconsistent "
                a1 = "meet yields "
                a2 = a1 ++ "morphism with inconsistent "
                a3 = "meet is not " in
            do s <- objs
               t <- objs
               let os = [s,t]
               f <- homset s t
               let fC = conv f
               let ms_C = [f,fC]
               (test (all_source c fC == t) os ms_C (c2 ++ "source") .
                test (all_target c fC == s) os ms_C (c2 ++ "target") .
                test (all_isMor c t s fC  ) os ms_C (c1 ++ "non-morphism") .
                let fCC = conv fC in
                test (fCC == f)    os (ms_C++[fCC]) (c1 ++ "no involution") .
                let f' = f &&& f in
                test (f == f')            os [f,f'] (a3 ++ "idempotent")
                ) : do
                 g <- homset s t
                 let gC = conv g
                let m = f &&& g
                let m' = g &&& f
                let ms = [f,g,m]
                let mC = conv m
                let cm = fC &&& gC
                (test (all_source c m == s) os ms (a2 ++ "source") .
                 test (all_target c m == t) os ms (a2 ++ "target") .
                 test (all_isMor c s t m)   os ms (a1 ++ "non-morphism") .
                 test (m == m')   os (ms ++ [m']) (a3 ++ "commutative") .
                 test ((f == m) == (f <<== g)) os ms
                      (a1 ++ "inclusion inconsistency") .
                 test (mC == cm) os (ms_C++[g,gC,mC,cm]) (convNPres ++ "meet")
--               test ((fC <<== gC) == (f <<== g)) os (ms_C++[gC])
--                    "non-monotone conversion"
                 ) : do
                  h <- homset s t
                  let m1 = m &&& h
                  let ms1 = [f,g,m,h,m1]
                  [let m2  =      g &&& h
                       m2' = f &&& m2      in
                   test (m2' == m1) os (ms1 ++ [m2,m2']) (a3 ++ "associative")
                   ]
                 ++ do
                  o3 <- objs
                  k <- homset o3 s
                  let kf = k ^ f
                  let kg = k ^ g
                  let km = k ^ m
                  let mk = kf &&& kg
                  [test (km <<== mk) (o3:os) [k,f,g,km,mk]
                        "meet-subdistributivity violated" ]
           ) .
     ffold (do o1 <- objs
               o2 <- objs
               o3 <- objs
               let os = [o1,o2,o3]
               g <- homset o1 o2
               let gC = conv g
               h <- homset o2 o3
               let gh = g ^ h
               let ghC = conv gh
               let hC = conv h
               let hCgC = hC ^ gC
               test (ghC == hCgC) os [g,h,ghC,hCgC] "converse is no functor" : do
                 f <- homset o1 o3
--               [test ((f &&& gh) <<== (((f ^ hC) &&& g) ^ ((gC ^ f) &&& h)))
--                     os [f,g,h] "Dedekind violation"]
                 [test ((f &&& gh) <<== (g ^ ((gC ^ f) &&& h)))
```

```
                       os [f,g,h] "violation of modal rule"]
           )
```

[FS90] define a *representation of allegories* to be a functor that preserves converse and meet — preservation of meet implies monotony.

Other sources, including [BDM97], define a *relator* to be a monotone functor between *tabular* allegories — there, monotony implies preservation of converse.

Since we are particularly interested in non-tabular allegories, we still employ the name of the latter, but define:

Definition 1.2.1 A *relator* is a monotone functor between allegories that preserves converse. A *representation of allegories* is a relator that also preserves meets. □

```
relator_TEST, allrepr_TEST :: Eq mor2 => All obj1 mor1 -> All obj2 mor2 ->
                                          Test (Fun obj2 mor2) obj1 mor1
relator_TEST = relator_TEST_frame False
allrepr_TEST = relator_TEST_frame True

relator_TEST_frame ::  Eq mor2 => Bool -> All obj1 mor1 -> All obj2 mor2 ->
                                           Test (Fun obj2 mor2) obj1 mor1
relator_TEST_frame allrepr c1 c2 fun =
  let objects1 = all_objects c1
      homset1 = all_homset c1
      (&&&) = all_meet c1
      (&&&&) = all_meet c2
      conv1 = all_converse c1
      conv2 = all_converse c2
      fo = fun_obj fun
      fm = fun_mor fun
      ident = if allrepr then "allegory representation" else "relator"
      message s = ident ++ " does not preserve " ++ s
  in ffold $ do
   s1 <- objects1
   t1 <- objects1
   let os = [s1,t1]
   f1 <- homset1 s1 t1
   let f2 = fm f1
   let f1C = conv1 f1
   let f2C = conv2 f2
   test (f2C == fm f1C) os [f1,f1C] (message "converse")
    : (do g1 <- homset1 s1 t1
          let g2 = fm g1
          if allrepr
           then let h1 = f1 &&& g1
                in [test (fm h1 == (f2 &&&& g2)) os [f1,g1,h1] (message "meet")]
           else let b = all_incl c2 f2 g2 || not (all_incl c1 f1 g1)
```

```
              in [test b os [f1,g1] (message "inclusion")]
      )
```

We shall sometimes need to test whether two allegories are equivalent; for this we assemble all relevant tests, creating a pair of `TestResults` of different types:

```
all_equiv_TESTS :: (Eq obj1, Eq mor1, Eq obj2, Eq mor2) =>
   All obj1 mor1 -> All obj2 mor2 ->
   Fun obj2 mor2 obj1 mor1 -> Fun obj1 mor1 obj2 mor2 ->
   (TestResult obj1  mor1, TestResult obj2 mor2)
all_equiv_TESTS a1 a2 f1 f2 =
  let c1 = all_cat a1
      c2 = all_cat a2
  in (functor_TEST c1 c2 f1 .
      allrepr_TEST a1 a2 f1 .
      functor_rightinv_test c1 c2 f2 f1
     ,functor_TEST c2 c1 f2 .
      allrepr_TEST a2 a1 f2 .
      functor_rightinv_test c2 c1 f1 f2
     )
```

This naïve procedure tends, however, to bind too much space; therefore we also define the corresponding sequence of `perform` actions:

```
all_equiv_perform a1 a2 f1 f2 =
  let c1 = all_cat a1
      c2 = all_cat a2
  in do perform (functor_TEST c1 c2) f1
        perform (functor_TEST c2 c1) f2
        perform (allrepr_TEST a1 a2) f1
        perform (allrepr_TEST a2 a1) f2
        perform (\ f1 -> functor_rightinv_test c1 c2 f2 f1) f1
        perform (\ f2 -> functor_rightinv_test c2 c1 f1 f2) f2
```

1.2.6 Distributive Allegories

Since all finite partial orders with a least element contain atoms, we include access to the atoms already in the interface of distributive allegories. Since this is a derived concept, we shall provide default definitions below.

```
data DistrAll obj mor = DistrAll
  {distrAll_all     :: All obj mor
  ,distrAll_bottom  :: obj -> obj -> mor
  ,distrAll_join    :: mor -> mor -> mor
  ,distrAll_atomset :: obj -> obj -> [mor]
  ,distrAll_atoms   :: mor -> [mor]
  }
```

We introduce an abbreviation that allows to directly access the bottom relation from the homset of a given morphism:

```
distrAll_bot da f = let s = distrAll_source da f
                        t = distrAll_target da f
                    in distrAll_bottom da s t


distrAll_isObj     = cat_isObj     . distrAll_cat   -- :: obj -> Bool
distrAll_isMor     = cat_isMor     . distrAll_cat   -- :: obj -> obj -> mor -> Bool
distrAll_objects   = cat_objects   . distrAll_cat   -- :: [obj]
distrAll_homset    = cat_homset    . distrAll_cat   -- :: obj -> obj -> [mor]
distrAll_source    = cat_source    . distrAll_cat   -- :: mor -> obj
distrAll_target    = cat_target    . distrAll_cat   -- :: mor -> obj
distrAll_idmor     = cat_idmor     . distrAll_cat   -- :: obj -> mor
distrAll_comp      = cat_comp      . distrAll_cat   -- :: mor -> mor -> mor

distrAll_cat       = all_cat       . distrAll_all
distrAll_converse  = all_converse  . distrAll_all   -- :: mor -> mor
distrAll_meet      = all_meet      . distrAll_all   -- :: mor -> mor -> mor
distrAll_incl      = all_incl      . distrAll_all   -- :: mor -> mor -> Bool
```

Testing distributive allegories is organised in the following way:

i) Two objects, one morphism: Bottom consistency inside one homset, idempotency of join

ii) Three objects one morphism: Zero law

iii) Two objects, two morphisms: Homset closed under join, commutativity and absorption laws

iv) Two objects, three morphisms: Associativity of join, lattice distributivity

v) Three objects, two morphisms: Distributivity of composition over join

```
distrAll_TEST :: (Eq obj, Eq mor) => Test DistrAll obj mor
distrAll_TEST c =
  let objects = distrAll_objects c
      (&&&) = distrAll_meet c
      (|||) = distrAll_join c
      (<<==) = distrAll_incl c
      (^) = distrAll_comp c
      homset = distrAll_homset c
      isMor = distrAll_isMor c
      bottom = distrAll_bottom c
      a1 = "join yields "
      a2 = a1 ++ "morphism with inconsistent "
```

```
    a3 = "join is not "
in ffold $ do
    s <- objects
    t <- objects
    let os = [s,t]
    let bot = bottom s t
    testX (isMor s t bot) [s,t] [bot] "bottom is non-morphism"
     $ do
      f <- homset s t
      let f' = f ||| f
      (test (bot <<== f) os [bot,f] "inconsistency of bottom wrt. inclusion" .
       test ((bot ||| f) == f) os [bot,f] "bottom not a unit for join" .
       test (f == f') os [f,f'] (a3 ++ "idempotent")
       ) : do
        u <- objects
        let botTU = bottom t u
        let botSU = bottom s u
        let fbot = f ^ botTU
        [test (fbot == botSU) os [f,botTU,fbot,botSU]
              "zero-law violated"                   ]
       ++ do
       g <- homset s t
       let j = f ||| g
       let j' = g ||| f
       let ms = [f,g,j]
       (test (distrAll_source c j == s) os ms (a2 ++ "source") .
        test (distrAll_target c j == t) os ms (a2 ++ "target") .
        test (isMor s t j)   os ms (a1 ++ "non-morphism") .
        test (j == j')   os (ms ++ [j']) (a3 ++ "commutative") .
        test (f &&& j == f) os ms "meet is not absorbing" .
        test (f ||| (f &&& g) == f) os ms (a3 ++ "absorbing")
        ) : do
         h <- homset s t
         let j1 = j ||| h
         let ms1 = [f,g,j,h,j1]
         let m1 = j &&& h
         let m2 = (f &&& h) ||| (g &&& h)
         [let j2  =       g ||| h
              j2' = f ||| j2      in
          test (j2' == j1) os (ms1++[j2,j2']) (a3 ++ "associative") .
          test (m1 == m2) os [f,g,h,m1,m2] "lattice not distributive"
          ]
        ++ do
         o3 <- objects
         k <- homset o3 s
         let kf = k ^ f
         let kg = k ^ g
         let kj = k ^ j
```

```
          let jk = kf ||| kg
          [test (kj == jk) (o3:os) [k,f,g,kj,jk] "join-distributivity violated"]
```

From the above test, together with the allegory test, it follows that $F \sqsubseteq G \Leftrightarrow F \sqcup G = G$, but this can also be tested separately:

```
distrAll_join_incl_TEST :: Eq mor => Test DistrAll obj mor
distrAll_join_incl_TEST c =
  let objects = distrAll_objects c
      (|||) = distrAll_join c
      (<<==) = distrAll_incl c
      homset = distrAll_homset c
  in ffold $ do
      s <- objects
      t <- objects
      f <- homset s t
      g <- homset s t
      let j = f ||| g
      [test ((j == g) == (f <<== g)) [s,t] [f,g]
            "inconsistency of join wrt. inclusion"]
```

A test for atomicity of a morphism only has to check all morphisms from the homset of the morphism in question for inclusion in that morphism:

```
distrAll_isAtom :: Eq mor => DistrAll obj mor -> obj -> obj -> mor -> Bool
distrAll_isAtom da s t m =
  let b = distrAll_bottom da s t
      homs = distrAll_homset da s t
      (<<==) = distrAll_incl da
  in distrAll_isMor da s t m &&
     m /= b &&
     all (\ m' -> (m' == m) || (m' == b) || not (m' <<== m)) homs
```

Filtering homsets with this test provides the default definition for atom lists:

```
distrAll_atomset_default all s t = filter (distrAll_isAtom all s t)
                                          (distrAll_homset all s t)
```

It is a fact that every finite Boolean lattice is atomic, i.e., every lattice element is the join of all atoms below it. Therefore, in a relation algebra the atom lists obtained by filtering the global atom sets with inclusion are, when considered as sets, a unique representation of the morphism in question. We provide these atom lists already here:

```
distrAll_atoms_default all m =
  let s = distrAll_source all m
      t = distrAll_target all m
  in filter (\ at -> distrAll_incl all at m) $ distrAll_atomset all s t
```

Our separate test for the atom components assumes an ordering on morphisms for being able to use a more efficient comparison:

```
distrAll_atomTEST :: Ord mor => Test DistrAll obj mor
distrAll_atomTEST da =
 let objects = distrAll_objects da
 in
 ffold $ do
   s <- objects
   t <- objects
   let os = [s,t]
   let atoms = distrAll_atomset_default da s t
   let atoms' = distrAll_atomset da s t
   test (atoms `listEqAsSet` atoms') os atoms' "inconsistent atom set" :
     do f <- distrAll_homset da s t
        let ats = distrAll_atoms_default da f
        let ats' = distrAll_atoms da f
        [test (ats `listEqAsSet` ats') os (f : ats')
              "inconsistent atom representation"]
```

1.2.7 Division Allegories

Division allegories only add three division operators:

```
data DivAll obj mor = DivAll
  {divAll_distrAll :: DistrAll obj mor
  ,divAll_rres :: mor -> mor -> mor
  ,divAll_lres :: mor -> mor -> mor
  ,divAll_syq  :: mor -> mor -> mor
  }
```

The symmetric quotient is defined on top of the residuals, which gives us the default definition:

```
divAll_syq_default :: DivAll obj mor -> mor -> mor -> mor
divAll_syq_default da f g = let conv = divAll_converse da in
   divAll_meet da (divAll_rres da f g)
                  (divAll_lres da (conv f) (conv g))
```

Each residual may be defined in terms of the other, i.e., $f \backslash g = (g\breve{}/f\breve{})\breve{}$ and $g/h = (h\breve{}\backslash g\breve{})\breve{}$:

```
divAll_rres_lresDefault :: DivAll obj mor -> mor -> mor -> mor
divAll_rres_lresDefault da f g =
   let conv = divAll_converse da
   in conv (divAll_lres da (conv g) (conv f))
```

```
divAll_lres_rresDefault :: DivAll obj mor -> mor -> mor -> mor
divAll_lres_rresDefault da g h =
   let conv = divAll_converse da
   in conv (divAll_rres da (conv h) (conv g))
```

For giving a default definition based on the residual properties, we need to be able to find the inclusion-maximal element of a set of morphisms; for this purpose we define the following auxiliary function:

```
poMax :: (a -> a -> Bool) -> a -> [a] -> a
--  Preconditions: ‘po‘ is a partial order
--                 bot  is the least element wrt. ‘po‘
--                 the list contains a maximal element
poMax po bot [] = bot
poMax po bot [x] = x
poMax po bot (x:y:ys)
  | x ‘po‘ y   =  poMax po y ys
  | y ‘po‘ x   =  poMax po x ys
  | otherwise  =  poMax po x ys -- Neither of x or y is the maximum,
                                -- but we do not try to catch errors.
                                -- Alternatively, we could use join here.
```

Translation of the residual specifications is now straightforward:

```
divAll_rres_inclDefault da f g =
   let (^) = divAll_comp da
       (<<==) = divAll_incl da
       target = divAll_target da
       s = target f
       t = target g
       ms = divAll_homset da s t
       check m = (f ^ m) <<== g
   in poMax (<<==) (divAll_bottom da s t) (filter check ms)
divAll_lres_inclDefault da g h =
   let (^) = divAll_comp da
       (<<==) = divAll_incl da
       source = divAll_source da
       s = source g
       t = source h
       ms = divAll_homset da s t
       check m = (m ^ h) <<== g
   in poMax (<<==) (divAll_bottom da s t) (filter check ms)
```

We provide separate tests for the different components, so that one may selectively test only those not defined via default definitions:

```
divAll_rres_TEST :: Test DivAll obj mor
divAll_rres_TEST da =
  let objects = divAll_objects da
      homset = divAll_homset da
      isMor = divAll_isMor da
      (^) = divAll_comp da
      (<<==) = divAll_incl da
      rres = divAll_rres da
  in ffold $ do
      s <- objects
      t <- objects
      g <- homset s t
      m <- objects
      f <- homset s m
      let r = f `rres` g
      testX (isMor m t r) [s,m,t] [f,g,r] "right residual yields non-morphism" $
       do
        h <- homset m t
        let fh = f ^ h
        [test ((h <<== r) == (fh <<== g)) [s,m,t] [g,f,r,h,fh]
              "right residual property violated"]
```

Although the corresponding test for left residuals textually differs only in minor points, the fact that these minor points affect the dependencies of the inner-most quantification and several places depending on it implies that factoring out the common parts would incur unreasonable costs in at least one of running time and readability.

```
divAll_lres_TEST :: Test DivAll obj mor
divAll_lres_TEST da =
  let objects = divAll_objects da
      homset = divAll_homset da
      isMor = divAll_isMor da
      (^) = divAll_comp da
      (<<==) = divAll_incl da
      lres = divAll_lres da
  in ffold $ do
      s <- objects
      t <- objects
      g <- homset s t
      m <- objects
      h <- homset m t
      let r = g `lres` h
      testX (isMor s m r) [s,m,t] [r,g,h] "left residual yields non-morphism" $
       do
        f <- homset s m
        let fh = f ^ h
        [test ((f <<== r) == (fh <<== g)) [s,m,t] [g,f,r,h,fh]
              "left residual property violated"]
```

The obvious and fast way to check the symmetric quotient is by verifying that its results correspond to the definition via residuals:

```
divAll_syq_resTEST :: Eq mor => Test DivAll obj mor
divAll_syq_resTEST da =
  let objects = divAll_objects da
      homset = divAll_homset da
      isMor = divAll_isMor da
      conv = divAll_converse da
      (&&&) = divAll_meet da
      lres = divAll_lres da
      rres = divAll_rres da
      syq = divAll_syq da
  in ffold $ do
      s <- objects
      m <- objects
      f <- homset m s
      t <- objects
      let os = [s,m,t]
      g <- homset m t
      let q = syq f g
      let l = rres f g
      let r = lres (conv f) (conv g)
      let ms = [f,g,q,l,r]
      testX (isMor s t q) os ms "syQ yields non-morphism"
        [test (q == (l &&& r)) os ms "syQ is not meet of residuals"]
```

However, the following definition of symmetric quotients also makes sense in the absence of residuals (see [FK98]):

Definition 1.2.2 In an allegory, the *symmetric quotient* $\mathrm{syq}(Q,S) : \mathcal{B} \leftrightarrow \mathcal{C}$ of two relations $Q : \mathcal{A} \leftrightarrow \mathcal{B}$ and $S : \mathcal{A} \leftrightarrow \mathcal{C}$ is defined by

$$X \sqsubseteq \mathrm{syq}(Q,S) \iff Q{;}X \sqsubseteq S \quad \text{and} \quad X{;}S^{\smile} \sqsubseteq Q^{\smile} \qquad \text{for all } X : \mathcal{B} \leftrightarrow \mathcal{C} \ . \qquad \square$$

This kind of symmetric quotient is of course usually a partial operation (it is obviously univalent). The following function calculates this operation:

```
all_syq :: All obj mor -> mor -> mor -> Maybe mor
all_syq a f g =
  let objects = all_objects a
      homset = all_homset a
      isMor = all_isMor a
      conv = all_converse a
      (^) = all_comp a
      (<<==) = all_incl a
      target = all_target a
      s = target f
      t = target g
      check q x = (x <<== q) == (((f ^ x) <<== g) && ((g ^ conv x) <<== f))
      syq q = all (check q) (homset s t)
   in listToMaybe $ filter syq $ homset s t
```

A given binary partial operation on morphisms can be tested for inclusion in this symmetric quotient by the following function:

```
all_syq_directTEST :: (mor -> mor -> Maybe mor) -> Test All obj mor
all_syq_directTEST syq a =
  let objects = all_objects a
      homset = all_homset a
      isMor = all_isMor a
      conv = all_converse a
      (^) = all_comp a
      (<<==) = all_incl a
  in ffold $ do
      s <- objects
      m <- objects
      f <- homset m s
      t <- objects
      let os = [s,m,t]
      g <- homset m t
      case syq f g of
        Nothing -> []
        Just q -> do
          let ms = [f,g,q]
          testX (isMor s t q) os ms "syQ yields non-morphism" $
           do x <- homset s  t
              let l = f ^      x <<==      g
              let r = x ^ conv g <<== conv f
              [test ((x <<== q) == (l && r)) os ms "syQ property violated"]
```

Finally, here is the definition of the expanded interface:

```
divAll_isObj    = cat_isObj        . divAll_cat   -- :: obj -> Bool
divAll_isMor    = cat_isMor        . divAll_cat   -- :: obj -> obj -> mor -> Bool
divAll_objects  = cat_objects      . divAll_cat   -- :: [obj]
divAll_homset   = cat_homset       . divAll_cat   -- :: obj -> obj -> [mor]
divAll_source   = cat_source       . divAll_cat   -- :: mor -> obj
divAll_target   = cat_target       . divAll_cat   -- :: mor -> obj
divAll_idmor    = cat_idmor        . divAll_cat   -- :: obj -> mor
divAll_comp     = cat_comp         . divAll_cat   -- :: mor -> mor -> mor

divAll_cat      = all_cat          . divAll_all
```

```
divAll_converse = all_converse     . divAll_all      -- :: mor -> mor
divAll_meet     = all_meet         . divAll_all      -- :: mor -> mor -> mor
divAll_incl     = all_incl         . divAll_all      -- :: mor -> mor -> Bool

divAll_all      = distrAll_all     . divAll_distrAll
divAll_bottom   = distrAll_bottom  . divAll_distrAll -- :: obj -> obj -> mor
divAll_bot      = distrAll_bot     . divAll_distrAll -- :: mor -> mor
divAll_join     = distrAll_join    . divAll_distrAll -- :: mor -> mor -> mor
divAll_atomset  = distrAll_atomset . divAll_distrAll -- :: obj -> obj -> [mor]
divAll_atoms    = distrAll_atoms   . divAll_distrAll -- :: mor -> [mor]
```

1.2.8 Dedekind Categories

```
data Ded obj mor = Ded
  {ded_divAll :: DivAll obj mor
  ,ded_top  :: obj -> obj -> mor
  }
```

Note that, as mentioned above, every finite distributive allegory is already a Dedekind category, so we can provide a default definition for `top`:

```
ded_top_default :: Ded obj mor -> obj -> obj -> mor
ded_top_default d s t = poMax (ded_incl d) (ded_bottom d s t) (ded_homset d s t)
```

In the same way as for bottom, we introduce an abbreviation that allows to directly access the top relation from the homset of a given morphism:

```
ded_tp ded f = let s = ded_source ded f
                   t = ded_target ded f
               in ded_top ded s t
```

The only item to test here is whether every morphism is indeed included in the top element of its homset:

```
ded_top_incl_TEST :: Eq mor => Test Ded obj mor
ded_top_incl_TEST c =
  let objects = ded_objects c
      top = ded_top c
      (<<==) = ded_incl c
      homset = ded_homset c
  in ffold $ do
      s <- objects
      t <- objects
      let tp = top s t
      test (ded_isMor c s t tp) [s,t] [tp] "top is non-morphism" : do
        f <- homset s t
        [test (f <<== tp) [s,t] [f,tp] "inconsistency of top wrt. inclusion"]
```

Finally, here is the expanded interface:

```
ded_isObj    = cat_isObj    . ded_cat       -- :: obj -> Bool
ded_isMor    = cat_isMor    . ded_cat       -- :: obj -> obj -> mor -> Bool
ded_objects  = cat_objects  . ded_cat       -- :: [obj]
ded_homset   = cat_homset   . ded_cat       -- :: obj -> obj -> [mor]
ded_source   = cat_source   . ded_cat       -- :: mor -> obj
ded_target   = cat_target   . ded_cat       -- :: mor -> obj
ded_idmor    = cat_idmor    . ded_cat       -- :: obj -> mor
ded_comp     = cat_comp     . ded_cat       -- :: mor -> mor -> mor

ded_cat      = all_cat      . ded_all
ded_converse = all_converse . ded_all       -- :: mor -> mor
ded_meet     = all_meet     . ded_all       -- :: mor -> mor -> mor
ded_incl     = all_incl     . ded_all       -- :: mor -> mor -> Bool

ded_distrAll = divAll_distrAll . ded_divAll

ded_all      = divAll_all      . ded_divAll
ded_bottom   = divAll_bottom   . ded_divAll -- :: obj -> obj -> mor
ded_bot      = divAll_bot      . ded_divAll -- :: mor -> mor
ded_join     = divAll_join     . ded_divAll -- :: mor -> mor -> mor
ded_atomset  = divAll_atomset  . ded_divAll -- :: obj -> obj -> [mor]
ded_atoms    = divAll_atoms    . ded_divAll -- :: mor -> [mor]
ded_rres     = divAll_rres     . ded_divAll -- :: mor -> mor -> mor
ded_lres     = divAll_lres     . ded_divAll -- :: mor -> mor -> mor
ded_syq      = divAll_syq      . ded_divAll -- :: mor -> mor -> mor
```

1.2.9 Relation Algebras

Not even every finite Dedekind category is a relation algebra, so the introduction of the complement is again a real step in advance:

```
data RA obj mor = RA
  {ra_ded   :: Ded obj mor
  ,ra_compl :: mor -> mor
  }
```

The presence of the complement allows more concise default definitions for the residuals:

```
ra_rres_default ra f g =
       let compl = ra_compl ra in compl (ra_comp ra (ra_converse ra f) (compl g))
ra_lres_default ra f g =
       let compl = ra_compl ra in compl (ra_comp ra (compl f) (ra_converse ra g))
```

Testing the complement is straightforward testing of the properties $F \sqcap \overline{F} = \perp\!\!\!\perp$ and $F \sqcup \overline{F} = \top\!\!\!\top$:

```
ra_compl_TEST :: Eq mor => Test RA obj mor
ra_compl_TEST c =
  let objects = ra_objects c
      homset = ra_homset c
      bot = ra_bottom c
      top = ra_top c
      not = ra_compl c
      (&&&) = ra_meet c
      (|||) = ra_join c
  in ffold $ do
      s <- objects
      t <- objects
      let tp = top s t
      let bt = bot s t
      f <- homset s t
      let fN = not f
      testX (ra_isMor c s t fN) [s,t] [fN] "complement yields non-morphism"
       (let m = f &&& fN
          j = f ||| fN in
      [test (m == bt) [s,t] [f,fN,m,bt] "meet with complement is not bottom" .
       test (j == tp) [s,t] [f,fN,j,tp] "join with complement is not top"
      ]
       )
```

For relation algebras with default definitions for all division operators the following is sufficient:

```
ra_TEST :: (Eq obj,Eq mor) => Test RA obj mor
ra_TEST ra =
  cat_TEST (ra_cat ra) .
  all_TEST (ra_all ra) .
  distrAll_TEST (ra_distrAll ra) .
  ded_top_incl_TEST (ra_ded ra) .
  ra_compl_TEST ra
```

Otherwise, there is also a variant with the atom tests (which require `Ord mor`) and the division tests included:

```
ra_TEST_ALL :: (Eq obj,Ord mor) => Test RA obj mor
ra_TEST_ALL ra =
  cat_TEST (ra_cat ra) .
  all_TEST (ra_all ra) .
  distrAll_TEST (ra_distrAll ra) .
  distrAll_atomTEST (ra_distrAll ra) .
  let da = ra_divAll ra in
  divAll_rres_TEST da .
  divAll_lres_TEST da .
  divAll_syq_resTEST da .
  ded_top_incl_TEST (ra_ded ra) .
  ra_compl_TEST ra
```

Finally, here is the expanded interface:

```
ra_isObj    = cat_isObj    . ra_cat      -- :: obj -> Bool
ra_isMor    = cat_isMor    . ra_cat      -- :: obj -> obj -> mor -> Bool
ra_objects  = cat_objects  . ra_cat      -- :: [obj]
ra_homset   = cat_homset   . ra_cat      -- :: obj -> obj -> [mor]
ra_source   = cat_source   . ra_cat      -- :: mor -> obj
ra_target   = cat_target   . ra_cat      -- :: mor -> obj
ra_idmor    = cat_idmor    . ra_cat      -- :: obj -> mor
ra_comp     = cat_comp     . ra_cat      -- :: mor -> mor -> mor

ra_cat      = all_cat      . ra_all
ra_converse = all_converse . ra_all      -- :: mor -> mor
ra_meet     = all_meet     . ra_all      -- :: mor -> mor -> mor
ra_incl     = all_incl     . ra_all      -- :: mor -> mor -> Bool

ra_all      = ded_all      . ra_ded
ra_bottom   = ded_bottom   . ra_ded      -- :: obj -> obj -> mor
ra_bot      = ded_bot      . ra_ded      -- :: mor -> mor
ra_join     = ded_join     . ra_ded      -- :: mor -> mor -> mor
ra_atomset  = ded_atomset  . ra_ded      -- :: obj -> obj -> [mor]
ra_atoms    = ded_atoms    . ra_ded      -- :: mor -> [mor]
ra_top      = ded_top      . ra_ded      -- :: obj -> obj -> mor
ra_tp       = ded_tp       . ra_ded      -- :: mor -> mor
ra_rres     = ded_rres     . ra_ded      -- :: mor -> mor -> mor
ra_lres     = ded_lres     . ra_ded      -- :: mor -> mor -> mor
ra_syq      = ded_syq      . ra_ded      -- :: mor -> mor -> mor

ra_divAll   = ded_divAll   . ra_ded
ra_distrAll = ded_distrAll . ra_ded
```

1.2.10 Simple Example Algebras

The following four algebras are not studied because they are interesting themselves. Rather, we need them as coefficients of matrix algebra constructions. So they are defined in a very detailed way so as to be able to proceed smoothly to more complex structures.

Trivial Relation Algebras

The smallest relation algebra has just one object and one morphism. Sometimes the definition of relation algebras requires that homsets be non-trivial Boolean lattices, but, as already mentioned, we do not follow this here.

Since we want to be able to talk about *embedded* relation algebras, we do not fix the object and morphism type. Instead, we build a trivial relation algebra from arbitrary objects and morphisms, as long as their types allow equality to be tested.

```
cat1 :: (Eq obj, Eq mor) => obj -> mor -> Cat obj mor
cat1 obj mor = Cat
  {cat_isObj   = (obj ==)
  ,cat_isMor   = const $ const $ (mor ==)
  ,cat_objects = [obj]
  ,cat_homset  = const $ const [mor]
  ,cat_source  = const obj
  ,cat_target  = const obj
  ,cat_idmor   = const mor
  ,cat_comp    = const $ const mor
  }


all1 :: (Eq obj, Eq mor) => obj -> mor -> All obj mor
all1 obj mor = All
  {all_cat = cat1 obj mor
  ,all_converse = id
  ,all_meet = const $ const mor
  ,all_incl = const $ const True
  }


distrAll1 :: (Eq obj, Eq mor) => obj -> mor -> DistrAll obj mor
distrAll1 obj mor = DistrAll
  {distrAll_all  = all1 obj mor
  ,distrAll_bottom = const $ const mor
  ,distrAll_join = const $ const mor
  ,distrAll_atomset = const $ const []
  ,distrAll_atoms = const []
  }


divAll1 :: (Eq obj, Eq mor) => obj -> mor -> DivAll obj mor
divAll1 obj mor = DivAll
  {divAll_distrAll  = distrAll1 obj mor
  ,divAll_rres = const $ const mor
  ,divAll_lres = const $ const mor
  ,divAll_syq  = const $ const mor
  }


ded1 :: (Eq obj, Eq mor) => obj -> mor -> Ded obj mor
ded1 obj mor = Ded
  {ded_divAll  = divAll1 obj mor
  ,ded_top  = const $ const mor
  }
```

```
ra1 :: (Eq obj, Eq mor) => obj -> mor -> RA obj mor
ra1 obj mor = RA
  {ra_ded = ded1 obj mor
  ,ra_compl = id
  }
```

Two-Element Relation Algebras

In the same way, we may define two-element relation algebras, where one morphism is bottom and the other morphism is identity and top at the same time:

```
cat2 :: (Eq obj, Eq mor) => obj -> mor -> mor -> Cat obj mor
cat2 obj bot id = Cat
  {cat_isObj   = (obj ==)
  ,cat_isMor   = const $ const $ (\ mor -> bot == mor || id == mor)
  ,cat_objects = [obj]
  ,cat_homset  = const $ const [bot,id]
  ,cat_source  = const obj
  ,cat_target  = const obj
  ,cat_idmor   = const id
  ,cat_comp    = (\ f g -> if f == id then g else bot)
  }


all2 :: (Eq obj, Eq mor) => obj -> mor -> mor -> All obj mor
all2 obj bot i = let c2 = cat2 obj bot i
 in All
  {all_cat = c2
  ,all_converse = id
  ,all_meet = cat_comp c2
  ,all_incl   = (\ f g -> f == bot || g == i)
  }


distrAll2 :: (Eq obj, Eq mor) => obj -> mor -> mor -> DistrAll obj mor
distrAll2 obj bot i = DistrAll
  {distrAll_all  = all2 obj bot i
  ,distrAll_bottom = const $ const bot
  ,distrAll_join = (\ f g -> if f == bot then g else i)
  ,distrAll_atomset = const $ const [i]
  ,distrAll_atoms   = (\ f -> if f == i then [i] else [])
  }


divAll2 :: (Eq obj, Eq mor) => obj -> mor -> mor -> DivAll obj mor
divAll2 obj bot i = DivAll
  {divAll_distrAll = distrAll2 obj bot i
  ,divAll_rres = (\ f g -> if f == bot || g == i   then i else bot)
  ,divAll_lres = (\ f g -> if f == i   || g == bot then i else bot)
  ,divAll_syq  = (\ f g -> if f == g               then i else bot)
  }


ded2 :: (Eq obj, Eq mor) => obj -> mor -> mor -> Ded obj mor
ded2 obj bot i = Ded
  {ded_divAll  = divAll2 obj bot i
  ,ded_top  = const $ const i
  }


ra2 :: (Eq obj, Eq mor) => obj -> mor -> mor -> RA obj mor
ra2 obj bot i = RA
  {ra_ded = ded2 obj bot i
  ,ra_compl = (\ f -> if f == bot then i else bot)
  }
```

The Relation Algebra $\mathbb{B}$

The relation algebra $\mathbb{B}$ of truth values might now be defined as `cat2 () False True`. For efficiency's sake, we also give a direct definition:

```
catB :: Cat () Bool
catB = Cat
  {cat_isObj   = const True
  ,cat_isMor   = const $ const $ const True
  ,cat_objects = [()]
  ,cat_homset  = const $ const [False, True]
  ,cat_source  = const ()
  ,cat_target  = const ()
  ,cat_idmor   = const True
  ,cat_comp    = (&&)
  }


allB :: All () Bool
allB = All
  {all_cat = catB
  ,all_converse = id
  ,all_meet = (&&)
  ,all_incl   = (\ f g -> g || not f)
  }


distrAllB :: DistrAll () Bool
distrAllB = DistrAll
  {distrAll_all  = allB
  ,distrAll_bottom = const $ const False
  ,distrAll_join = (||)
  ,distrAll_atomset = const $ const [True]
  ,distrAll_atoms   = (\ f -> if f then [True] else [])
  }


divAllB :: DivAll () Bool
divAllB = DivAll
  {divAll_distrAll = distrAllB
  ,divAll_rres = (\ f g -> not f || g)
  ,divAll_lres = (\ f g -> not g || f)
  ,divAll_syq  = (==)
  }


dedB :: Ded () Bool
dedB = Ded
  {ded_divAll  = divAllB
  ,ded_top  = const $ const True
  }


raB :: RA () Bool
raB = RA
  {ra_ded = dedB
  ,ra_compl = not
  }
```

$(n+1)$-Element Linearly Ordered Dedekind Categories

We now give a set of examples of “discretely fuzzy” Dedekind categories which are not relation algebras. As in $\mathbb{B}$, there is only one object and the identity is the maximum morphism, but there is a linearly ordered set of morphisms below the identity. For simplicity, we use initial segments `[0 .. n]` of the natural numbers as homsets.

Composition coincides with meet and is the minimum:

```
catN :: Eq obj => obj -> Int -> Cat obj Int
catN obj n = Cat
  {cat_isObj   = (obj ==)
  ,cat_isMor   = const $ const $ (\ k -> 0 <= k && k <= n)
  ,cat_objects = [obj]
  ,cat_homset  = const $ const [0 .. n]
  ,cat_source  = const obj
  ,cat_target  = const obj
  ,cat_idmor   = const n
  ,cat_comp    = min
  }
```

Conversion is the identity function on morphisms:

```
allN :: Eq obj => obj -> Int -> All obj Int
allN obj n = All
  {all_cat = catN obj n
  ,all_converse = id
  ,all_meet = min
  ,all_incl   = (<=)
  }
```

Join is of course maximum, and the only atom is 1:

```
distrAllN :: Eq obj => obj -> Int -> DistrAll obj Int
distrAllN obj n = da where
 da = DistrAll
  {distrAll_all  = allN obj n
  ,distrAll_bottom = const $ const 0
  ,distrAll_join = max
  ,distrAll_atomset = (\ s t -> if n > 0 then [1] else [])
  ,distrAll_atoms   = (\ f   -> if f > 0 then [1] else [])
  }
```

For residuals, we use the defaults:

```
divAllN :: Eq obj => obj -> Int -> DivAll obj Int
divAllN obj n = da where
 da = DivAll
  {divAll_distrAll = distrAllN obj n
  ,divAll_rres = divAll_rres_inclDefault da
  ,divAll_lres = divAll_lres_inclDefault da
  ,divAll_syq  = divAll_syq_default da
  }
```

Maximum morphisms are trivial again:

```
dedN :: Eq obj => obj -> Int -> Ded obj Int
dedN obj n = Ded
  {ded_divAll  = divAllN obj n
  ,ded_top  = const $ const n
  }
```

Just for fun, we also define a relation algebra constructor with a bogus complement function which will only work for $n \in \{0, 1\}$.

```
raN :: Eq obj => obj -> Int -> RA obj Int
raN obj n = RA
  {ra_ded = dedN obj n
  ,ra_compl = (n -)
  }
```

Consequently, `ra_TEST_ALL (raN () 2)` fails in `ra_compl_TEST`, exhibiting the middle morphism which has no complement.

1.3 Properties and Interesting Configurations

We start the definition of tests with some very simple ones, since it is important to look for all the details, too.

```
module Properties where

import RelAlg
```

1.3.1 Simple Morphism Properties

The following tests for the categorical definition of monomorphisms:

```
cat_isMono :: Eq mor => Cat obj mor -> mor -> Bool
cat_isMono c m = noResults (cat_mono_TEST m) c

cat_mono_TEST :: Eq mor => mor -> Test Cat obj mor
cat_mono_TEST h c =
  let objects = cat_objects c
      homset = cat_homset c
      (^) = cat_comp c
      s = cat_source c h
      t = cat_target c h
  in ffold $ do
   a <- objects
   let homs = homset a s
   f <- homs
   g <- homs
   let fh = f ^ h
   let gh = g ^ h
   [test ((fh == gh) == (f == g)) [a,s,t] [f,g,h,fh,gh] "mono counterexample"]
```

The dual then tests for epimorphisms:

```
cat_isEpi :: Eq mor => Cat obj mor -> mor -> Bool
cat_isEpi c m = noResults (cat_epi_TEST m) c

cat_epi_TEST :: Eq mor => mor -> Test Cat obj mor
cat_epi_TEST h c =
  let objects = cat_objects c
      homset = cat_homset c
      (^) = cat_comp c
      s = cat_source c h
      t = cat_target c h
  in ffold $ do
   c <- objects
   let homs = homset t c
   f <- homs
   g <- homs
   let hf = h ^ f
   let hg = h ^ g
   [test ((hf == hg) == (f == g)) [s,t,c] [h,f,g,hf,hg] "epi counterexample"]
```

In allegories, we already have the usual relational definitions of univalence, totality, injectivity, and surjectivity:

```
all_univalent_TEST m a =
  let mC = all_converse a m
      t = all_target a m
      iT = all_idmor a t
      mCm = all_comp a mC m
  in test (all_incl a mCm iT) [all_source a m, t] [m,mC,mCm,iT] "not univalent"

all_injective_TEST m a =
  let mC = all_converse a m
      s = all_source a m
      iS = all_idmor a s
      mmC = all_comp a m mC
  in test (all_incl a mmC iS) [s, all_target a m] [m,mC,mmC,iS] "not injective"

all_total_TEST m a =
  let mC = all_converse a m
      s = all_source a m
      iS = all_idmor a s
      mmC = all_comp a m mC
  in test (all_incl a iS mmC) [s, all_target a m] [m,mC,mmC,iS] "not total"

all_surjective_TEST m a =
  let mC = all_converse a m
      t = all_target a m
      iT = all_idmor a t
      mCm = all_comp a mC m
  in test (all_incl a iT mCm) [all_source a m, t] [m,mC,mCm,iT] "not surjective"
```

Of these, we also provide Boolean variants:

```
all_isUnivalent  a m = noResults (all_univalent_TEST  m) a
all_isInjective  a m = noResults (all_injective_TEST  m) a
all_isTotal      a m = noResults (all_total_TEST      m) a
all_isSurjective a m = noResults (all_surjective_TEST m) a
all_isMapping    a m = all_isUnivalent a m && all_isTotal a m
```

Sometimes it may be interesting what the non-trivial mappings in an allegory are; here we offer an accordingly restricted version of `all_homset` and a function that collects all *non-trivial* mappings of an allegory into a `TestResult`:

```
all_mappings a s t = let
 in filter (all_isMapping a) $ all_homset a s t

all_mapTest :: (Eq obj, Eq mor) => Test All obj mor
all_mapTest a = let
    objects = all_objects a
 in ffold $ do
     s <- objects
     t <- objects
     let noId = if s /= t then id else let i = all_idmor a s in filter (/= i)
     let ms = noId $ all_mappings a s t
     case ms of [] -> []
                _ -> [test False [s,t] ms "mappings"]
```

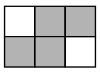
The same can be done for functions; since we consider not only identities, but also empty relations as trivial functions, the test has a separate variant for distributive allegories:

```
all_functions a s t = let
 in filter (all_isUnivalent a) $ all_homset a s t

all_funTest :: (Eq obj, Eq mor) => Test All obj mor
all_funTest a = let
    objects = all_objects a
 in ffold $ do
     s <- objects
     t <- objects
     let noId = if s /= t then id else let i = all_idmor a s in filter (/= i)
     let ms = noId $ all_functions a s t
     case ms of [] -> []
                _ -> [test False [s,t] ms "functions"]

distrAll_funTest :: (Eq obj, Eq mor) => Test DistrAll obj mor
distrAll_funTest a = let
    objects = distrAll_objects a
 in ffold $ do
     s <- objects
     t <- objects
     let noId = if s /= t then id
                          else let i = distrAll_idmor a s in filter (/= i)
     let ms = filter (/= (distrAll_bottom a s t)) $ noId
              $ all_functions (distrAll_all a) s t
     case ms of [] -> []
                _ -> [test False [s,t] ms "functions"]
```

If $R : \mathcal{A} \leftrightarrow \mathcal{B}$ is injective and total, then $R{;}R^{\smile} = \mathbb{I}_A$, so R is obviously mono. It is also trivial that every mono has to be total. But it is not so easy to see that, in general, not every mono has to be injective, so we write a quick test:

```
all_mono_inj_TEST a =
  let objects = all_objects a
      homset = all_homset a
      isMono m = cat_isMono (all_cat a) m
      isTot m = all_isTotal a m
      isInj m = all_isInjective a m
  in ffold $ do
   s <- objects
   t <- objects
   m <- homset s t
   let mono = isMono m
   let inj = isInj m
   [test (mono <= inj) [s,t] [m] "mono, but not injective"]
```

A monomorphism which is not injective is the following Boolean 2×3-matrix:



This is one of the smallest Boolean matrices with this property; in most of the relation algebras of the third chapter, all monomorphisms are injective.

1.3.2 Homogeneous Relations

We also provide a few tests for frequently-used properties of homogeneous relations (all without checking for homogeneity):

```
all_reflexive_TEST m a =
  let s = all_source a m
      iS = all_idmor a s
  in test (all_incl a iS m) [s] [m,iS] "not reflexive"

all_coreflexive_TEST m a =
  let s = all_source a m
      iS = all_idmor a s
  in test (all_incl a m iS) [s] [m,iS] "not coreflexive"

all_symmetric_TEST m a =
  let mC = all_converse a m
  in test (all_incl a m mC) [all_source a m] [m,mC] "not symmetric"

all_transitive_TEST m a =
  let mm = all_comp a m m
  in test (all_incl a mm m) [all_source a m] [m,mm] "not transitive"

all_antisymmetric_TEST m a =
  let mC = all_converse a m
      x = all_meet a m mC
      s = all_source a m
      iS = all_idmor a s
  in test (all_incl a x iS) [s] [m,mC,x,iS] "not antisymmetric"


all_preorder_TEST    m a = all_reflexive_TEST m a . all_transitive_TEST m a
all_order_TEST       m a = all_preorder_TEST  m a . all_antisymmetric_TEST m a
all_equivalence_TEST m a = all_preorder_TEST  m a . all_symmetric_TEST m a


all_isReflexive     a m = noResults (all_reflexive_TEST     m) a
all_isCoreflexive   a m = noResults (all_coreflexive_TEST   m) a
all_isSymmetric     a m = noResults (all_symmetric_TEST     m) a
all_isTransitive    a m = noResults (all_transitive_TEST    m) a
all_isAntisymmetric a m = noResults (all_antisymmetric_TEST m) a
all_isOrder         a m = noResults (all_order_TEST         m) a
all_isPreorder      a m = noResults (all_preorder_TEST      m) a
all_isEquivalence   a m = noResults (all_equivalence_TEST   m) a
```

1.3.3 Uniformity

Definition 1.3.1 A Dedekind category is called *uniform* if for all objects $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$ we have

$$\mathbb{T}_{\mathcal{A},\mathcal{B}};\mathbb{T}_{\mathcal{B},\mathcal{C}} = \mathbb{T}_{\mathcal{A},\mathcal{C}} \qquad \square$$

In heterogeneous relation algebras, uniformity is implied by the Tarski rule. It is, however, cheaper to test:

```
ded_uniform_TEST :: Eq mor => Test Ded obj mor
ded_uniform_TEST d =
  let objects = ded_objects d
      top = ded_top d
  in ffold $ do
       o1 <- objects
       o2 <- objects
       let t12 = top o1 o2
       o3 <- objects
       let t23 = top o2 o3
       let t13 = top o1 o3
       let t = ded_comp d t12 t23
       [test (t == t13) [o1,o2,o3] [t12,t23,t13,t] "non-uniform"]
```

1.3.4 Units

According to [FS90, 2.15]:

Definition 1.3.2 An object $\mathcal{U}$ in an allegory is a *partial unit* if $\mathbb{I}_{\mathcal{U}}$ is its maximum endomorphism. $\mathcal{U}$ is a *unit* if, further, every object is the source of a total morphism targeted at $\mathcal{U}$. An allegory is said to be *unitary* if it has a unit. □

Testing for partial units can be done in allegories, but is (usually) much more efficient in Dedekind categories where there is immediate access to the maximum morphisms:

```
all_partialUnit_TEST :: obj -> Test All obj mor
all_partialUnit_TEST u a =
  let iU = all_idmor a u
      (<<==) = all_incl a
  in ffold (do m <- all_homset a u u
               [test (m <<== iU) [u] [iU,m] "identity is not maximal"])

ded_partialUnit_TEST :: Eq mor => obj -> Test Ded obj mor
ded_partialUnit_TEST u a =
  let iU = ded_idmor a u
      tU = ded_top a u u
  in (test (iU == tU) [u] [iU,tU] "identity is not maximal")
```

For the unit test, we first of all need a totality test:

```
ded_isTotal d = all_isTotal (ded_all d)
```

Given a partial unit, we can test whether it is a unit with the following test:

```
all_partialUnit_unit_TEST :: obj -> Test All obj mor
all_partialUnit_unit_TEST u a =                -- Precondition: u is partial unit
  let objects = all_objects a
      homset = all_homset a
      check s = any (all_isTotal a) (homset s u)
  in ffold $ do
      s <- objects
      [test (check s) [s,u] [] "no total morphism to unit"]
```

We integrate this test directly into the unit search functions; because of the different complexity of the partial unit test we again provide this function both for allegories and for Dedekind categories:

```
all_units :: All obj mor -> [obj]
all_units a =
  let objects = all_objects a
```

```
      homset = all_homset a
      check u s = any (all_isTotal a) (homset s u)
      punit u = noResults (all_partialUnit_TEST u) a
      unit u = punit u && all (check u) objects
  in filter unit objects

ded_units :: Eq mor => Ded obj mor -> [obj]
ded_units a =
  let objects = ded_objects a
      homset = ded_homset a
      check u s = any (ded_isTotal a) (homset s u)
      punit u = noResults (ded_partialUnit_TEST u) a
      unit u = punit u && all (check u) objects
  in filter unit objects
```

1.3.5 Tabulations

According to [FS90], a pair f, g of maps *tabulates* a morphism R iff

$$f^{\smile};g = R \qquad \text{and} \qquad f;f^{\smile} \sqcap g;g^{\smile} = \mathbb{I} \ .$$

Actually, it is sufficient to demand that f and g be univalent, since the second condition implies their totality.

The heart of the tabulation test therefore has the following precondition: $R : s \leftrightarrow t$, $f : p \nrightarrow s$, $g : p \nrightarrow t$.

```
is_tabulation :: (Eq mor) => All obj mor ->
                             obj -> obj -> mor ->
                             obj -> mor -> mor -> TestResult obj mor
is_tabulation a s t r p f g =
  let (^) = all_comp a
      (&&&) = all_meet a
      conv = all_converse a
      os = [s,t,p]
      fC = conv f
      gC = conv g
      fCg = fC ^ g
      cc = (f ^ fC) &&& (g ^ gC)
      ip = all_idmor a p
  in test (fCg == r) os [r,f,g,fCg]   "tabulation not correct"
   . test (cc == ip) os [r,f,g,cc,ip] "tabulation not precise"
```

Since the non-standard algebras we are looking for are certainly *not* tabular, we do not provide a test for tabularity.

1.3.6 Direct Products

It is well-known that the self-duality of categories of relations implies that categorical sums are at the same time categorical products — in relation algebras with sets and concrete relations, categorical sums are disjoint unions.

However, Cartesian products can be axiomatised appropriately on the relational level [ZSB86, SS93]:

Definition 1.3.3 A *direct product* for two objects $\mathcal{A}$ and $\mathcal{B}$ is a triple $(\mathcal{P}, \pi, \rho)$ consisting of an object $\mathcal{P}$ and two *projections*, i.e., relations $\pi : \mathcal{P} \leftrightarrow \mathcal{A}$ and $\rho : \mathcal{P} \leftrightarrow \mathcal{B}$ for which the following conditions hold:

$$\pi\breve{}\,;\pi = \mathbb{I} \;, \qquad \rho\breve{}\,;\rho = \mathbb{I} \;, \qquad \pi\breve{}\,;\rho = \mathbb{T} \;, \qquad \pi;\pi\breve{} \sqcap \rho;\rho\breve{} = \mathbb{I} \;. \qquad \square$$

In our product data type, we explicitly mention all three objects involved:

```
type Product obj mor = (obj,obj,obj,mor,mor)
```

The last two conditions for direct products are equivalent to saying that the projections tabulate $\mathbb{T}_{\mathcal{A},\mathcal{B}}$, and we use this in our test:

```
ded_isNonemptyProduct :: (Eq obj, Eq mor) =>
    obj -> obj -> obj -> mor -> mor ->  Test Ded obj mor
ded_isNonemptyProduct a b p pA pB d =
  let alleg = ded_all d
      source = ded_source d
      target = ded_target d
      top = ded_top d
      (^) = ded_comp d
--      (&&&) = ded_meet d
      conv = ded_converse d
      pAC = conv pA
      pBC = conv pB
      idmor = ded_idmor d
      iA = idmor a
      iB = idmor b
--      iP = idmor p
  in
  test (source pA == p) [p,a] [pA] "inconsistent source of first projection" .
  test (target pA == a) [p,a] [pA] "inconsistent target of first projection" .
  test (source pB == p) [p,b] [pB] "inconsistent source of second projection" .
  test (target pB == a) [p,b] [pB] "inconsistent target of second projection" .
  test (pAC ^ pA == iA) [a,p] [pA,pAC,iA]
       "first projection not {univalent and surjective}" .
  test (pBC ^ pB == iB) [b,p] [pB,pBC,iB]
```

```
      "second projection not {univalent and surjective}" .
  is_tabulation alleg a b (top a b) p pA pB
--test (pAC ^ pB == top a b) [a,p,b] [pAC,pB] "non-comprehensive product" .
--test ((pA ^ pAC) &&& (pB ^ pBC) == iP) [a,p,b] [pA,pAC,pB,pBC,iP]
--     "product not {univalent and total}"
```

For the sake of speed, we use an integrated version of these tests when searching for products; we also demand an ordering on objects and only return products over pairs of objects inside that ordering:

```
ded_NonemptyProducts :: (Eq obj, Ord obj, Eq mor) =>
                        Ded obj mor -> [Product obj mor]
ded_NonemptyProducts d =
  let objects = ded_objects d
      homset = ded_homset d
      top = ded_top d
      (^) = ded_comp d
      (&&&) = ded_meet d
      conv = ded_converse d
      idmor = ded_idmor d
  in do p <- objects
        let iP = idmor p
        a <- objects
        let iA = idmor a
        pA <- homset p a
        let pAC = conv pA
        if pAC ^ pA /= iA then []
         else do
          b <- objects
          let iB = idmor b
          pB <- homset p b
          let pBC = conv pB
          if b < a || pBC ^ pB /= iB then []
           else
            if (pAC ^ pB == top a b)
            && ((pA ^ pAC) &&& (pB ^ pBC) == iP)
            then [(a,b,p,pA,pB)]
            else []
```

A simpler variant only checks whether two projections can be found for a given triple of objects:

```
ded_NonemptyProducts1 :: (Eq obj, Eq mor) => obj -> obj -> obj ->
                                             Ded obj mor -> [Product obj mor]
ded_NonemptyProducts1 a b p d =
  let homset = ded_homset d
      top = ded_top d
      (^) = ded_comp d
      (&&&) = ded_meet d
      conv = ded_converse d
      idmor = ded_idmor d
      iP = idmor p
      iA = idmor a
      iB = idmor b
  in do pA <- homset p a
        let pAC = conv pA
        if pAC ^ pA /= iA then []
         else do
          pB <- homset p b
          let pBC = conv pB
          if pBC ^ pB /= iB then []
           else
            if (pAC ^ pB == top a b)
            && ((pA ^ pAC) &&& (pB ^ pBC) == iP)
            then [(a,b,p,pA,pB)]
            else []
```

For all direct products in relation algebras, the following inclusion holds:

$$P{;}R \sqcap Q{;}S \sqsupseteq (P{;}\pi^{\smile} \sqcap Q{;}\rho^{\smile}){;}(\pi{;}R \sqcap \rho{;}S).$$

The opposite inclusion

$$P{;}R \sqcap Q{;}S \sqsubseteq (P{;}\pi^{\smile} \sqcap Q{;}\rho^{\smile}){;}(\pi{;}R \sqcap \rho{;}S)$$

does not always hold. It is, however, trivial to prove it in the context of relations in the classical sense. The inability to prove it relation-algebraically first came up in 1981, when Rodrigo Cardoso prepared his diploma thesis [Car82] under the supervision of the second-named author who convinced himself that this might indeed be impossible, who named it the *sharpness condition*, and who conjectured that there might be “unsharp” models of relation algebra.

For a relation algebra with an unsharp product, together with its history, see Sect. 3.2. Since the search for computationally relevant models with unsharp products constitutes a main motivation for our current endeavour, we need a test whether a given product is unsharp:

```
ded_unsharp ::  (Eq obj, Eq mor) => Product obj mor -> Test Ded obj mor
ded_unsharp (a,b,_,pA,pB) d =
  let objects = ded_objects d
      homset = ded_homset d
      (^) = ded_comp d
      (&&&) = ded_meet d
      conv = ded_converse d
      pAC = conv pA
      pBC = conv pB
  in ffold (do
    x <- objects
    xA <- homset x a
    let xAP = xA ^ pAC
    xB <- homset x b
    let xBP = xB ^ pBC
    let xP = xAP &&& xBP
    y <- objects
    aY <- homset a y
    let xAY = xA ^ aY
    let pAY = pA ^ aY
    bY <- homset b y
    let xBY = xB ^ bY
    let pBY = pB ^ bY
    let pY = pAY &&& pBY
    let xPY = xP ^ pY
    let xY = xAY &&& xBY
    [test (xPY == xY) [x,y] [xA,xB,aY,bY,xPY,xY] "unsharpness example"]
   )
```

1.3.7 Standard Iterations

There is a well-developed theory of standard iterations for boolean matrices to be found along with matching, assignment, games, correctness, etc. We will present tools for executing these iterations in the general framework studied here.

To this end, we concentrate on pairs of antitone mappings occurring in relation algebras. These mappings are usually determined by an obviously antitone relational construct, e.g., $w \mapsto \pi(w) := \overline{B;w}$ based on the relation $B : V \leftrightarrow W$. Many other antitone mappings are conceivable.

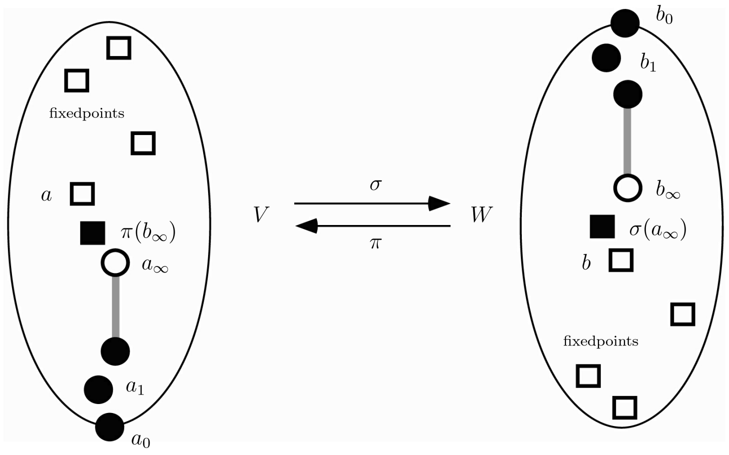
Such pairs lead to interesting Galois correspondences of their fixed-points, and give rise to iteration procedures. From the numerous application areas studied with relations in the classical sense we mention the following:

	given relations	$\sigma(v) =$	$\pi(w) =$
bi-matrix games	$B : W \leftrightarrow V, B'\ V \leftrightarrow W$	$\overline{B;v}$	$\overline{B';w}$
correctness	$B : V \leftrightarrow V$	$B;\overline{v}$	$\overline{w}$
minorants, majorants	$E : V \leftrightarrow V$	$\overline{\overline{E};w}$	$\overline{\overline{E^\smile};v}$
coverings	$Q : V \leftrightarrow W$	$Q^\smile;\overline{v}$	$Q;\overline{w}$
independence	$Q : V \leftrightarrow W$	$\overline{Q^\smile;v}$	$\overline{Q;w}$
assignment	$\lambda \sqsubseteq Q : V \leftrightarrow W$	$\overline{Q^\smile;v}$	$\overline{\lambda;w}$

The antitone mappings are related to study questions such as the following:

$B;\overline{q} \sqsubseteq \overline{q}$	contraction	partial correctness
$\overline{q} \sqsubseteq B;\overline{q}$	complement expansion	total correctness
$B;x \sqsubseteq \overline{x}$	stability	kernels, games
$\overline{x} \sqsubseteq B;x$	absorption	

Some of these classical iterations are presented in [SS85a, SS93]. Here, they are schematically transferred into the present setting.



Bounds and fixed-points of antitone mappings

We express σ, π with the elementary operations of the respective algebra and apply them to an appropriate starting configuration a_0, b_0. Appropriate means that $a_0 \sqsubseteq \pi(b_0) \sqsubseteq a$ and $b \sqsubseteq \sigma(a_0) \sqsubseteq b_0$, where a is the least fixed-point of $\rho = \lambda v.\pi(\sigma(v))$ and b is the greatest fixed-point of $\psi = \lambda w.\sigma(\pi(w))$. This complicated condition is usually satisfied rather trivially with $a_0 = \perp\!\!\!\perp$ and $b_0 = \top\!\!\!\top$. Nested iterations will then start with a_0 on the left and b_0 on the right:

$$a_{i+1} := \pi(b_i) \ , \qquad b_{i+1} := \sigma(a_i) \ .$$

These iterations will end up in the following two sequences, one of which is ascending while the other descends.

$$a_0 \sqsubseteq a_1 \sqsubseteq \ldots \sqsubseteq a_\infty \sqsubseteq \pi(b_\infty) \sqsubseteq a, \quad b \sqsubseteq \sigma(a_\infty) \sqsubseteq b_\infty \sqsubseteq \ldots \sqsubseteq b_1 \sqsubseteq b_0.$$

The effect of the iteration is that the least fixed-point a of $v \mapsto \pi(\sigma(v))$ on the side started with a_0 is related to the greatest fixed-point b of $w \mapsto \sigma(\pi(w))$ on the side started from b_0. The final situation obtained will be characterised by $a = \pi(b)$ and $\sigma(a) = b$. It will always produce another admissible starting configuration $a'_0 := a_\infty, b'_0 := b_\infty$.

It is not clear from the beginning whether the iteration will reach the fixed-points in a finite number of steps, as the mapping $w \mapsto \pi(w)$, e.g., might not be continuous. In interesting applications, however, this is the case; in particular in the finite case. (Of course this iteration may also be executed the other way round, i.e., starting with $\top\!\!\!\top$ on the left and with $\perp\!\!\!\perp$ on the right.)

The configuration is more specific in the homogeneous case. The sequences may meet each other or may fail to do so.

In any case, interesting investigations have been possible in the case of relations in the classical sense. It is challenging to look for interpretations of similar results in the context of the more general examples of relation algebras presented here.

The program module for standard iterations starts with a module heading.

```
module Iterations where

import RelAlg
import Matrix
import ExtPrel
```

Then the basic iterations along the well-known `until`-construct of Haskell with `lr` for σ and `rl` for π are formulated.

```
antiFix :: (Eq a, Eq b) => (b -> a) -> (a -> b) -> (b,a) -> (b,a)
antiFix lr rl = untilFix f
  where f (v,w) = (rl w, lr v)
```

For reasons of monotony, the iteration will always terminate at the fixed-points in the finite case; see [SS93, A.3.11].

The start may often be determined from the row and column number of the given basic relations inserting $\perp\!\!\!\perp$ and $\top\!\!\!\top$ as appropriate.

```
startVector :: MatMor obj mor -> (obj -> obj -> mor') -> MatMor obj mor'
startVector b tf = let (m,s,t) = unMatMor b
                       t' = head s
                       m' = map ((:[]) . flip tf t') t
                   in MatMor (m',t,[t'])
```

We will now apply this general scheme to several applications.

Initial Part

A nice example for a fixed-point of antitone functionals is determining the initial part of a relation along the lines of [SS93, 6.3.4]. (Remember, however, that for nonfinite relations the concepts of being progressively finite and progressively bounded will be different.) The two antitone functionals $v \mapsto B;\overline{v}$ and $w \mapsto \overline{w}$ are given as follows:

```
antitoneFctlCorr1 ra b = \ v -> ra_comp ra b $ ra_compl ra v
antitoneFctlCorr2 ra b = \ w -> ra_compl ra w          -- independent of b!
```

Applying the general scheme, we obtain the initial part in the resulting pair of

```
initialPart ra b = antiFix (antitoneFctlCorr1 ra b)
                           (antitoneFctlCorr2 ra b)
                           (startVector b ccFalse, startVector b ccTrue)
```

Bi-Matrix Games

Next, we look for solutions of bi-matrix games. Let two matrices $B : V \leftrightarrow W, B' : W \leftrightarrow V$ be given. The antitone functionals based on these relations are formed in quite a similar manner.

```
antitonFctlGame ra = \ b -> (\ x -> ra_compl ra (ra_comp ra b x))
```

The solution of the game is then again determined following the general scheme.

```
gameSolution ra b b' = antiFix (antitonFctlGame ra b )
                                (antitonFctlGame ra b')
                                (startVector b' ccTrue, startVector b ccFalse)
gameSolutINV ra b b' = antiFix (antitonFctlGame ra b )
                                (antitonFctlGame ra b')
                                (startVector b' ccFalse, startVector b ccTrue)
```

The final situation is characterised by the formulae $a = \overline{B;b}$ and $\overline{B';a} = b$ for the game iteration as well as for the inverted iteration. The respective smaller resulting relation gives loss positions, while the larger ones indicate loss positions together with draw positions.

Matching and Assignment

An additional antimorphism situation is known to exist in connection with matching and assignment. Let two matrices $Q, \lambda : V \leftrightarrow W$ be given where $\lambda \sqsubseteq Q$ is univalent.

```
antitoneFctlAssign ra = \ b -> (\ x -> ra_compl ra (ra_comp ra b x))

assignSolution ra q lambda
      = antiFix (antitoneFctlAssign ra (ra_converse ra q))
                (antitoneFctlAssign ra lambda)
                (startVector q ccFalse
                ,startVector (ra_converse ra lambda) ccTrue)

assignSolutINV ra q lambda
      = antiFix (antitoneFctlAssign ra (ra_converse ra q))
                (antitoneFctlAssign ra lambda)
                (startVector q ccTrue
                ,startVector (ra_converse ra lambda) ccFalse)
```

In the classical case, the results of these iterations produce appropriate starting points, if any, where to apply successfully the alternating chain procedure.

Bounds wrt. Orderings

The following functions offer the possibility to calculate the majorants and the minorants of a relation as well as least upper and greatest lower bounds, provided the corresponding ordering is given as a first argument.

```
mi, ma, lub, glb :: DivAll obj mor -> mor -> mor -> mor

mi da ord m = divAll_lres da ord (divAll_converse da m)
ma da ord m = let conv = divAll_converse da
              in divAll_lres da (conv ord) (conv m)

lub da ord m = divAll_meet da maom (mi da ord maom)  where maom = ma da ord m
glb da ord m = divAll_meet da miom (ma da ord miom)  where miom = mi da ord m
```

This might also be done using the `antiFix`-scheme presented in this section. From theoretical considerations, however, it is clear that these iterations will be stationary after one step back and forth. This makes a direct computation the better choice.

Conclusion

It seems extremely interesting, to find out how these standard iterations behave if matrices are taken the coefficients of which are drawn from other relation algebras. Do, e.g., matrices over an interval algebra lead to steering algorithms? Will game algorithms over matrices with pairs (interval, compass) give hints to escape games? Will there be targeting games?

1.4 Interoperability With the Class Interface

Although the class interface of Sect. 1.1 and the dictionary records of Sect. 1.2 are completely independent of each other, it is easy to obtain interoperability between the two interfaces.

In this section we first instantiate the classes of Sect. 1.1 for the dictionary types of Sect. 1.2 in 1.4.1. We then show in 1.4.2 how to obtain explicit dictionaries from class interfaces, and apply this to transfer the test functions defined in Sect. 1.2 from the dictionary setting to the class setting in 1.4.3.

```
module RelAlgInstances where

import RelAlg
import RelAlgClasses
import Properties
import Atomset
```

1.4.1 Instantiating the Class Interface

Categories

```
instance Category (Cat obj mor) obj mor where
  isObj   = cat_isObj
  isMor   = cat_isMor
  objects = cat_objects
  homset  = cat_homset
  source  = cat_source
  target  = cat_target
  idmor   = cat_idmor
  comp    = cat_comp

instance Category (All obj mor) obj mor where
  isObj   = all_isObj
  isMor   = all_isMor
  objects = all_objects
  homset  = all_homset
  source  = all_source
  target  = all_target
  idmor   = all_idmor
  comp    = all_comp

instance Category (DistrAll obj mor) obj mor where
  isObj   = distrAll_isObj
  isMor   = distrAll_isMor
  objects = distrAll_objects
  homset  = distrAll_homset
  source  = distrAll_source
  target  = distrAll_target
  idmor   = distrAll_idmor
  comp    = distrAll_comp

instance Category (DivAll obj mor) obj mor where
  isObj   = divAll_isObj
  isMor   = divAll_isMor
  objects = divAll_objects
  homset  = divAll_homset
  source  = divAll_source
  target  = divAll_target
  idmor   = divAll_idmor
  comp    = divAll_comp

instance Category (Ded obj mor) obj mor where
  isObj   = ded_isObj
  isMor   = ded_isMor
  objects = ded_objects
  homset  = ded_homset
  source  = ded_source
  target  = ded_target
  idmor   = ded_idmor
  comp    = ded_comp

instance Category (RA obj mor) obj mor where
  isObj   = ra_isObj
  isMor   = ra_isMor
  objects = ra_objects
  homset  = ra_homset
  source  = ra_source
  target  = ra_target
  idmor   = ra_idmor
  comp    = ra_comp
```

Allegories

```
instance Allegory (All obj mor) obj mor where
  converse = all_converse
  meet     = all_meet
  incl     = all_incl

instance Allegory (DistrAll obj mor) obj mor where
  converse = distrAll_converse
  meet     = distrAll_meet
  incl     = distrAll_incl

instance Allegory (DivAll obj mor) obj mor where
  converse = divAll_converse
  meet     = divAll_meet
  incl     = divAll_incl

instance Allegory (Ded obj mor) obj mor where
  converse = ded_converse
  meet     = ded_meet
  incl     = ded_incl

instance Allegory (RA obj mor) obj mor where
  converse = ra_converse
  meet     = ra_meet
  incl     = ra_incl
```

Distributive Allegories

```
instance DistribAllegory (DistrAll obj mor) obj mor where
  join   = distrAll_join
  bottom = distrAll_bottom
```

```
instance DistribAllegory (DivAll obj mor) obj mor where
  join   = divAll_join
  bottom = divAll_bottom

instance DistribAllegory (Ded obj mor) obj mor where
  join   = ded_join
  bottom = ded_bottom

instance DistribAllegory (RA obj mor) obj mor where
  join   = ra_join
  bottom = ra_bottom
```

Division Allegories

```
instance DivisionAllegory (DivAll obj mor) obj mor where
  rres = divAll_rres
  lres = divAll_lres
  syq  = divAll_syq

instance DivisionAllegory (Ded obj mor) obj mor where
  rres = ded_rres
  lres = ded_lres
  syq  = ded_syq

instance DivisionAllegory (RA obj mor) obj mor where
  rres = ra_rres
  lres = ra_lres
  syq  = ra_syq
```

Dedekind Categories

```
instance DedCat (Ded obj mor) obj mor where
  top  = ded_top

instance DedCat (RA obj mor) obj mor where
  top  = ra_top
```

Relation Algebras

```
instance RelAlg (RA obj mor) obj mor where
  compl = ra_compl
```

1.4.2 Reverse Instances

```
catDict :: Category cat obj mor => cat -> Cat obj mor
catDict c = Cat
```

```
  {cat_isObj   = isObj c
  ,cat_isMor   = isMor c
  ,cat_objects = objects c
  ,cat_homset  = homset c
  ,cat_source  = source c
  ,cat_target  = target c
  ,cat_idmor   = idmor c
  ,cat_comp    = comp c
  }
```

```
allDict :: Allegory all obj mor => all -> All obj mor
allDict a = All
  {all_cat = catDict a
  ,all_converse = converse a
  ,all_meet = meet a
  ,all_incl = incl a
  }
```

```
distrAllDict :: (DistribAllegory da obj mor, Eq mor) => da -> DistrAll obj mor
distrAllDict da = da' where
 da' = DistrAll
  {distrAll_all  = allDict da
  ,distrAll_bottom = bottom da
  ,distrAll_join = join da
  ,distrAll_atomset = distrAll_atomset_default da'
  ,distrAll_atoms   = distrAll_atoms_default da'
  }
```

```
divAllDict :: (DivisionAllegory da obj mor, Eq mor) => da -> DivAll obj mor
divAllDict da = DivAll
  {divAll_distrAll = distrAllDict da
  ,divAll_rres = rres da
  ,divAll_lres = lres da
  ,divAll_syq  = syq da
  }
```

```
dedDict :: (DedCat ded obj mor, Eq mor) => ded -> Ded obj mor
dedDict ded = Ded
  {ded_divAll  = divAllDict ded
  ,ded_top  = top ded
  }
```

```
raDict :: (RelAlg ra obj mor, Eq mor) => ra -> RA obj mor
raDict ra = RA
  {ra_ded = dedDict ra
  ,ra_compl = compl ra
  }


acatDict ::(DistribAllegory da obj mor, Ord obj, Eq mor) => da -> ACat obj mor
acatDict = distrAll_acat . distrAllDict

aallDict ::(DistribAllegory da obj mor, Ord obj, Eq mor) => da -> AAll obj mor
aallDict = distrAll_aall . distrAllDict
```

1.4.3 Transfer of Tests

Using these “dictionary explicators”, we can lift our testing machinery to the class setting:

```
category_TEST ::
  (Category cat obj mor, Eq obj, Eq mor) => cat -> TestResult obj mor
category_TEST = cat_TEST . catDict

allegory_TEST ::
  (Allegory all obj mor, Eq obj, Eq mor) => all -> TestResult obj mor
allegory_TEST = all_TEST . allDict

distribAllegory_TEST ::
  (DistribAllegory da obj mor, Eq obj, Eq mor) => da -> TestResult obj mor
distribAllegory_TEST = distrAll_TEST . distrAllDict

distribAllegory_join_incl_TEST ::
  (DistribAllegory da obj mor, Eq obj, Eq mor) => da -> TestResult obj mor
distribAllegory_join_incl_TEST = distrAll_join_incl_TEST . distrAllDict

divisionAllegory_rres_TEST, divisionAllegory_lres_TEST
 , divisionAllegory_syq_resTEST
   :: (DivisionAllegory da obj mor, Eq obj, Eq mor) => da -> TestResult obj mor
divisionAllegory_rres_TEST   = divAll_rres_TEST   . divAllDict
divisionAllegory_lres_TEST   = divAll_lres_TEST   . divAllDict
divisionAllegory_syq_resTEST = divAll_syq_resTEST . divAllDict

allegory_syq_directTEST :: (Allegory all obj mor, Eq obj, Eq mor) =>
   (mor -> mor -> Maybe mor) -> all -> TestResult obj mor
allegory_syq_directTEST syq = all_syq_directTEST syq . allDict

dedCat_top_incl_TEST ::
  (DedCat ded obj mor, Eq obj, Eq mor) => ded -> TestResult obj mor
dedCat_top_incl_TEST = ded_top_incl_TEST . dedDict
```

```
relAlg_compl_TEST, relAlg_TEST
   :: (RelAlg ra obj mor, Eq obj, Eq mor) => ra -> TestResult obj mor
relAlg_compl_TEST = ra_compl_TEST . raDict
relAlg_TEST       = ra_TEST       . raDict

relAlg_TEST_ALL
   :: (RelAlg ra obj mor, Eq obj, Ord mor) => ra -> TestResult obj mor
relAlg_TEST_ALL   = ra_TEST_ALL   . raDict
```

This makes the following direct queries possible:

```
HugsMain> perform category_TEST   ra_McKenzie
No results.

HugsMain> perform allegory_TEST   ra_LRNnoc
No results.

HugsMain> perform relAlg_TEST_ALL ra_Winter
No results.
```

We also transfer other tests:

```
uniform_TEST :: (DedCat ded obj mor, Eq obj, Eq mor) => ded -> TestResult obj mor
uniform_TEST = ded_uniform_TEST . dedDict

allegory_partialUnit_TEST :: (Allegory all obj mor, Eq obj, Eq mor) =>
  obj -> all -> TestResult obj mor
allegory_partialUnit_TEST u = all_partialUnit_TEST u . allDict

dedCat_partialUnit_TEST :: (DedCat all obj mor, Eq obj, Eq mor) =>
  obj -> all -> TestResult obj mor
dedCat_partialUnit_TEST u = ded_partialUnit_TEST u . dedDict

allegory_partialUnit_unit_TEST :: (Allegory all obj mor, Eq obj, Eq mor) =>
  obj -> all -> TestResult obj mor
allegory_partialUnit_unit_TEST u = all_partialUnit_unit_TEST u . allDict

dedCat_units :: (DedCat ded obj mor, Eq obj, Eq mor) => ded -> [obj]
dedCat_units = ded_units . dedDict

isNonemptyProduct :: (DedCat ded obj mor, Eq obj, Eq mor) =>
                     obj -> obj -> obj -> mor -> mor -> ded -> TestResult obj mor
isNonemptyProduct oA oB oP pA pB = ded_isNonemptyProduct oA oB oP pA pB . dedDict

nonemptyProducts ::
  (DedCat ded obj mor, Ord obj, Eq mor) => ded -> [Product obj mor]
nonemptyProducts = ded_NonemptyProducts . dedDict
```

```
nonemptyProducts1 :: (DedCat ded obj mor, Eq obj, Eq mor) =>
                     obj -> obj -> obj -> ded -> [Product obj mor]
nonemptyProducts1 oA oB oP = ded_NonemptyProducts1 oA oB oP . dedDict

unsharp :: (DedCat ded obj mor, Eq obj, Eq mor) =>
           Product obj mor -> ded -> TestResult obj mor
unsharp p = ded_unsharp p . dedDict
```

Now we can directly formulate queries such as in the following session:

```
HugsMain> nonemptyProducts ra_Maddux
[(B,C,A,SetMor ({At1},A,B),SetMor ({At1},A,C))]
HugsMain> performAll (unsharp (head $ nonemptyProducts ra_Maddux)) ra_Maddux
=== Test Start ===
unsharpness example
 Objects:
  D
  E
 Morphisms:
  SetMor ({At1},D,B)
  SetMor ({At1},D,C)
  SetMor ({At1},B,E)
  SetMor ({At1},C,E)
  SetMor ({At2},D,E)
  SetMor ({At1, At2},D,E)
unsharpness example
 Objects:
  E
  D
 Morphisms:
  SetMor ({At1},E,B)
  SetMor ({At1},E,C)
  SetMor ({At1},B,D)
  SetMor ({At1},C,D)
  SetMor ({At2},E,D)
  SetMor ({At1, At2},E,D)
=== Test End   ===
```

Chapter 2

Relation Algebra Construction

With tools of today such as RelView, we are able to exhaustively handle all the relations on a 5-element set or between a 4-element and a 6-element set on a computer, e.g. As there are $2^{A \times B}$ relations between sets A and B, we should no pretend to be able to handle all these relations appropriately in the same way. A closer look, however, makes clear, that much less relations are being under consideration, namely those composed by union, intersection, and composition of "rectangular" basic blocks. These in turn stem from conditions on the first resp. on the second component of a pair. So the way a single relation from a relation algebra is constructed deserves further study.

We investigate, therefore, product algebras, sub-algebras, matrix algebras, etc.

2.1 Product Algebras

The construction of product algebras, where all operations are defined component-wise, is completely straightforward, only requiring an appropriate set of pair lifting functions (`prodF`, `prodFF`, `cprodFF`, `prodFFF`, `cprodFFF`) all defined in Sect. A.3. We therefore do not need to comment on the individual steps of the construction.

```
module Product where

import ExtPrel
import RelAlg



catProd :: Cat obj1 mor1 -> Cat obj2 mor2 -> Cat (obj1,obj2) (mor1,mor2)
catProd c1 c2 = Cat
  {cat_isObj   = pairAnd  . prodF    (cat_isObj   c1)   (cat_isObj   c2)
  ,cat_isMor   = pairAnd  `cprodFFF` (cat_isMor   c1) $ (cat_isMor   c2)
  ,cat_objects = listProd            (cat_objects c1,    cat_objects c2)
  ,cat_homset  = listProd `cprodFF`  (cat_homset  c1) $ (cat_homset  c2)
  ,cat_source  =            prodF    (cat_source  c1)   (cat_source  c2)
  ,cat_target  =            prodF    (cat_target  c1)   (cat_target  c2)
  ,cat_idmor   =            prodF    (cat_idmor   c1)   (cat_idmor   c2)
  ,cat_comp    =            prodFF   (cat_comp    c1)   (cat_comp    c2)
  }


allProd :: All obj1 mor1 -> All obj2 mor2 -> All (obj1,obj2) (mor1,mor2)
allProd c1 c2 = All
```

```
  {all_cat      = catProd (all_cat c1) (all_cat c2)
  ,all_converse =          prodF   (all_converse c1)   (all_converse c2)
  ,all_meet     =          prodFF  (all_meet     c1)   (all_meet     c2)
  ,all_incl     = pairAnd `cprodFF` (all_incl     c1) $ (all_incl     c2)
  }



distrAllProd :: DistrAll obj1 mor1 -> DistrAll obj2 mor2 ->
                DistrAll (obj1,obj2) (mor1,mor2)
distrAllProd c1 c2 = let mkAts1 b = map (\a -> (a,b))
                         mkAts2 b = map (\a -> (b,a))
 in DistrAll
  {distrAll_all     = allProd (distrAll_all c1) (distrAll_all c2)
  ,distrAll_bottom  = prodFF (distrAll_bottom  c1) (distrAll_bottom  c2)
  ,distrAll_join    = prodFF (distrAll_join    c1) (distrAll_join    c2)
  ,distrAll_atomset = (\ (s1,s2) (t1,t2) ->
       mkAts1 (distrAll_bottom c2 s2 t2) (distrAll_atomset c1 s1 t1)
    ++ mkAts2 (distrAll_bottom c1 s1 t1) (distrAll_atomset c2 s2 t2))
  ,distrAll_atoms   = (\ (f1,f2) ->
       mkAts1 (distrAll_bot    c2 f2   ) (distrAll_atoms   c1 f1   )
    ++ mkAts2 (distrAll_bot    c1 f1   ) (distrAll_atoms   c2 f2   ))
  }



divAllProd :: DivAll obj1 mor1 -> DivAll obj2 mor2 ->
              DivAll (obj1,obj2) (mor1,mor2)
divAllProd c1 c2 = DivAll
  {divAll_distrAll = distrAllProd (divAll_distrAll c1) (divAll_distrAll c2)
  ,divAll_rres     = prodFF (divAll_rres c1) (divAll_rres c2)
  ,divAll_lres     = prodFF (divAll_lres c1) (divAll_lres c2)
  ,divAll_syq      = prodFF (divAll_syq  c1) (divAll_syq  c2)
  }



dedProd :: Ded obj1 mor1 -> Ded obj2 mor2 -> Ded (obj1,obj2) (mor1,mor2)
dedProd c1 c2 = Ded
  {ded_divAll = divAllProd (ded_divAll c1) (ded_divAll c2)
  ,ded_top    = prodFF (ded_top c1) (ded_top c2)
  }



raProd :: RA obj1 mor1 -> RA obj2 mor2 -> RA (obj1,obj2) (mor1,mor2)
raProd c1 c2 = RA
  {ra_ded   = dedProd (ra_ded c1) (ra_ded c2)
  ,ra_compl = prodF (ra_compl c1) (ra_compl c2)
  }
```

2.2 Sub-Algebras

Forming sub-algebras is another standard algebra construction mechanism. It is particularly promising in connection with relation algebras, as it is known that one may sometimes take a subset of all the available relations and will still maintain the basic structure.

```
module SubAlg where

import FiniteMaps
import Sets
import ExtPrel
import RelAlg
```

Our approach is to use an auxiliary `SubCat` data structure to contain the information necessary to define a sub-algebra of a given algebra. From the mathematical point of view, this additional information consists of the object set and of the homsets of the sub-algebra; all operations are preserved.

Using Sub-Algebras

This information can then be used to obtain a sub-algebra from an algebra for all kinds of algebras under consideration in this report:

```
sub_cat      :: (Ord o, Ord m) => SubCat o m -> Cat      o m -> Cat      o m
sub_all      :: (Ord o, Ord m) => SubCat o m -> All      o m -> All      o m
sub_distrAll :: (Ord o, Ord m) => SubCat o m -> DistrAll o m -> DistrAll o m
sub_divAll   :: (Ord o, Ord m) => SubCat o m -> DivAll   o m -> DivAll   o m
sub_ded      :: (Ord o, Ord m) => SubCat o m -> Ded      o m -> Ded      o m
sub_ra       :: (Ord o, Ord m) => SubCat o m -> RA       o m -> RA       o m
```

Higher-level structures only add operations, so the only difference is in the underlying structure of the next lower level. In the following functions we always assume the `SubCat` structure to have been checked for closedness under the relevant operations and therefore omit tests:

```
sub_ra       s c = c {ra_ded          = sub_ded      s (ra_ded          c)}
sub_ded      s c = c {ded_divAll      = sub_divAll   s (ded_divAll      c)}
sub_divAll   s c = c {divAll_distrAll = sub_distrAll s (divAll_distrAll c)}
sub_distrAll s c = c {distrAll_all    = sub_all      s (distrAll_all    c)}
sub_all      s c = c {all_cat         = sub_cat      s (all_cat         c)}
```

For categories, the sub-category obtains redefined object and homset components, but inherits the unchanged operations:

```
sub_cat s@(SubCat objs hs) c = c
  {cat_isObj   = (\ o -> o `elemSet` objs)
  ,cat_isMor   = (\ a b m -> m `elemSet` lookupDftFM hs zeroSet (a,b))
  ,cat_objects = toListSet objs
  ,cat_homset  = curry (toListSet . lookupDftFM hs zeroSet)
  }
```

The preferred interface, however, takes an arbitrary `SubCat` data structure, closes it under the relevant operations, and applies the above functions to obtain the corresponding sub-algebra:

```
subCat       :: (Ord o, Ord m) => SubCat o m -> Cat      o m -> Cat      o m
subAll       :: (Ord o, Ord m) => SubCat o m -> All      o m -> All      o m
subDistrAll :: (Ord o, Ord m) => SubCat o m -> DistrAll o m -> DistrAll o m
subDivAll    :: (Ord o, Ord m) => SubCat o m -> DivAll   o m -> DivAll   o m
subDed       :: (Ord o, Ord m) => SubCat o m -> Ded      o m -> Ded      o m
subRA        :: (Ord o, Ord m) => SubCat o m -> RA       o m -> RA       o m
```

These functions are defined below, after introduction of the necessary machinery.

Sub-Algebra Closure Machinery

By introducing a very abstract and comprehensive interface to the `SubCat` data type, one might be able to define sub-algebras without resorting to even `Eq` instances for the object and morphism data types. However, we think this is not worth the effort. For the sake of efficiency, we even demand `Ord` instances and do not consider this as a serious restriction for the kind of uses we have in mind.

For `obj` and `mor` types in the `Ord` class we can directly implement the `SubCat` data type via standard set and finite map data structures[1]:

```
data SubCat obj mor = SubCat
  {sub_objects :: Set obj
  ,sub_homset :: FiniteMap (obj,obj) (Set mor)
  }
```

Simple lookup functions:

```
sub_isMor :: (Ord obj, Ord mor) => SubCat obj mor -> obj -> obj -> mor -> Bool
sub_isMor (SubCat objs mors) a b m =
   case lookupFM mors (a,b) of
     Nothing -> False
     Just mors -> m `elemSet` mors

sub_isEmpty (SubCat objs hs) = isZeroSet objs && isZeroFM hs
```

[1]These are imported from the modules `Sets` and `FiniteMaps` taken from Manuel Chakravarty's compiler toolkit and slightly modified for our purposes. We prefer this variant over those provided by GHC (from which they are derived) for portability reasons since they work with other Haskell implementations as well and do not give rise to name clashes with GHC.

Adding a single morphism to some homset:

```
addToHomset :: (Ord obj, Ord mor) => obj -> obj -> mor
                                        -> SubCat obj mor -> SubCat obj mor
addToHomset a b m (SubCat objs hs) = SubCat objs $ addToFM (a,b) mors' hs
 where mors' = case lookupFM hs (a,b) of
                 Nothing -> unitSet m
                 Just mors -> addToSet m mors
```

Joining two (intermediate) `SubCat` data structures:

```
subcat_join (SubCat objs1 hs1) (SubCat objs2 hs2) =
  SubCat (objs1 `joinSet` objs2) (foldFM f hs1 hs2)
 where f p ms hs = addToFM p ms' hs
        where ms' = case lookupFM hs p of
                      Nothing -> ms
                      Just mors -> mors `joinSet` ms


type SubCatDiff obj mor = SubCat obj mor

type SubCatClosure obj mor = STFun (SubCat obj mor) Bool
```

`SubCatClosure`s can be composed, yielding the conjunction of the intermediate results:

```
scComp :: (Ord obj, Ord mor) =>
           SubCatClosure obj mor -> SubCatClosure obj mor ->
           SubCatClosure obj mor
scComp f g = do b1 <- f
                b2 <- g
                return (b1 && b2)
```

`SubCatClosure`s will usually be created via `scStep` from a function calculating an incremental `SubCatDiff` from an intermediate `SubCat`:

```
scStep :: (Ord obj, Ord mor) =>
          (SubCat obj mor -> SubCatDiff obj mor) -> SubCatClosure obj mor
scStep f = STFun (\ s -> let d = f s
                             b = sub_isEmpty d
                             s' = subcat_join s d
                         in (s', b))
```

Applying a `SubCatClosure` means iterating it until the incremental difference is empty:

```
scClose :: (Ord obj, Ord mor) =>
           SubCatClosure obj mor -> SubCat obj mor -> SubCat obj mor
scClose step s = fst $ applySTFun iter s
  where iter = do b <- step
                  if b then return () else iter
```

Sub-Algebra Closure Functions

After thus establishing the machinery, we now present the individual difference creation functions; these are then used by the sub-algebra generators.

The simplest closure is creating a sub-category induced by a set of objects; this only has to take all morphisms between those objects and needs not be iterated:

```
cat_homset_closeSubCatDiff :: (Ord obj, Ord mor) =>
                          Cat obj mor -> SubCat obj mor -> SubCatDiff obj mor
cat_homset_closeSubCatDiff c s =
  let objects = toListSet $ sub_objects s
      idmor = cat_idmor c
      homset = cat_homset c
      adds = do a <- objects
                b <- objects
                f <- homset a b
                if sub_isMor s a b f then []
                                     else [addToHomset a b f]
  in foldr id (SubCat zeroSet zeroFM) adds

cat_homset_close :: (Ord obj, Ord mor) =>
                          Cat obj mor -> SubCat obj mor -> SubCat obj mor
cat_homset_close c s = s `subcat_join` cat_homset_closeSubCatDiff c s
```

A more dedicated function could eliminate the cost of morphism lookup, which is logarithmic in the sizes of the object set and of the homset in question. However, we postpone this until it is felt to be a bottle neck.

For turning arbitrary `SubCat` data structures into legal sub-category descriptions, we first of all have to make sure that all identities are present:

```
cat_id_closeSubCatDiff :: (Ord obj, Ord mor) =>
                          Cat obj mor -> SubCat obj mor -> SubCatDiff obj mor
cat_id_closeSubCatDiff c s =
  let objects = toListSet $ sub_objects s
      idmor = cat_idmor c
      notthere o i = not (sub_isMor s o o i)
  in foldSet (\ o r -> let i = idmor o in if notthere o i
                       then addToHomset o o i r
                       else r)
             (SubCat zeroSet zeroFM)
             (sub_objects s)
```

Next we close the homsets under composition:

```
cat_comp_closeSubCatDiff :: (Ord obj, Ord mor) =>
                          Cat obj mor -> SubCat obj mor -> SubCatDiff obj mor
```

```
cat_comp_closeSubCatDiff c s@(SubCat objs hs) =
  let objects = toListSet $ sub_objects s
      homset a b = toListSet $ lookupDftFM hs zeroSet (a,b)
      (^) = cat_comp c
      comps = do a <- objects
                 b <- objects
                 f <- homset a b
                 c <- objects
                 g <- homset b c
                 let h = f ^ g
                 if sub_isMor s a c h then []
                                      else [addToHomset a c h]
  in foldr id (SubCat zeroSet zeroFM) comps
```

These two are sufficient for sub-categories:

```
cat_closeStep :: (Ord obj, Ord mor) => Cat obj mor -> SubCatClosure obj mor
cat_closeStep c = scStep (cat_id_closeSubCatDiff c) ‘scComp‘
                  scStep (cat_comp_closeSubCatDiff c)

subCat s c = sub_cat (scClose (cat_closeStep c) s) c
```

For allegories, we have to close under conversion and meet:

```
all_conv_closeSubCatDiff :: (Ord obj, Ord mor) =>
                          All obj mor -> SubCat obj mor -> SubCatDiff obj mor
all_conv_closeSubCatDiff c s@(SubCat objs hs) =
  let objects = toListSet $ sub_objects s
      homset a b = toListSet $ lookupDftFM hs zeroSet (a,b)
      conv = all_converse c
      (&&&) = all_meet c
      convs = do a <- objects
                 b <- objects
                 f <- homset a b
                 let g = conv f
                 (if sub_isMor s b a g then id
                                       else ((addToHomset b a g)  :))
                  $ do
                   g <- homset a b
                   let h = f &&& g
                   if sub_isMor s a b h then []
                                        else [addToHomset a b h]
  in foldr id (SubCat zeroSet zeroFM) convs


all_closeStep :: (Ord obj, Ord mor) => All obj mor -> SubCatClosure obj mor
all_closeStep c = cat_closeStep (all_cat c) ‘scComp‘
                  scStep (all_conv_closeSubCatDiff c)

subAll s c = sub_all (scClose (all_closeStep c) s) c
```

For distributive allegories, we simultaneously add bottom morphisms and close under joins:

```
distrAll_closeSubCatDiff :: (Ord obj, Ord mor) =>
                         DistrAll obj mor -> SubCat obj mor -> SubCatDiff obj mor
distrAll_closeSubCatDiff c s@(SubCat objs hs) =
  let objects = toListSet $ sub_objects s
      homset a b = toListSet $ lookupDftFM hs zeroSet (a,b)
      bot = distrAll_bottom c
      (|||) = distrAll_join c
      adds = do a <- objects
                b <- objects
                let t = bot a b
                (if sub_isMor s a b t then id else ((addToHomset a b t) :))
                 $ do
                  f <- homset a b
                  g <- homset a b
                  let h = f ||| g
                  if sub_isMor s a b h then []
                                      else [addToHomset a b h]
  in foldr id (SubCat zeroSet zeroFM) adds


distrAll_closeStep :: (Ord obj, Ord mor) =>
                      DistrAll obj mor -> SubCatClosure obj mor
distrAll_closeStep c = all_closeStep (distrAll_all c) `scComp`
                       scStep (distrAll_closeSubCatDiff c)

subDistrAll s c = sub_distrAll (scClose (distrAll_closeStep c) s) c
```

For division allegories, we only have to add left and right residuals — symmetric quotients are added as intersections of those in the allegory step:

```
divAll_closeSubCatDiff :: (Ord obj, Ord mor) =>
                          DivAll obj mor -> SubCat obj mor -> SubCatDiff obj mor
divAll_closeSubCatDiff c s@(SubCat objs hs) =
  let objects = toListSet $ sub_objects s
      homset a b = toListSet $ lookupDftFM hs zeroSet (a,b)
      lres = divAll_lres c
      rres = divAll_rres c
      comps = do a <- objects
                 b <- objects
                 f <- homset a b
                 c <- objects
                 (do g <- homset a c
                     let h = g `rres` f
                     if sub_isMor s b c h then []
                                         else [addToHomset b c h]
                  ) ++ (do
```

```
                    g <- homset c b
                    let h = f `lres` g
                    if sub_isMor s a b h then []
                                          else [addToHomset a b h]
                )
  in foldr id (SubCat zeroSet zeroFM) comps


divAll_closeStep :: (Ord obj, Ord mor) => DivAll obj mor -> SubCatClosure obj mor
divAll_closeStep c = distrAll_closeStep (divAll_distrAll c) `scComp`
                     scStep (divAll_closeSubCatDiff c)

subDivAll s c = sub_divAll (scClose (divAll_closeStep c) s) c
```

For Dedekind categories, we only need to add top morphisms:

```
ded_closeSubCatDiff :: (Ord obj, Ord mor) =>
                       Ded obj mor -> SubCat obj mor -> SubCatDiff obj mor
ded_closeSubCatDiff c s@(SubCat objs hs) =
  let objects = toListSet $ sub_objects s
      homset a b = toListSet $ lookupDftFM hs zeroSet (a,b)
      top = ded_top c
      adds = do a <- objects
                b <- objects
                let t = top a b
                if sub_isMor s a b t then [] else [addToHomset a b t]
  in foldr id (SubCat zeroSet zeroFM) adds


ded_closeStep :: (Ord obj, Ord mor) => Ded obj mor -> SubCatClosure obj mor
ded_closeStep c = divAll_closeStep (ded_divAll c) `scComp`
                  scStep (ded_closeSubCatDiff c)

subDed s c = sub_ded (scClose (ded_closeStep c) s) c
```

Complementation is the only operation we have to check for relation algebras:

```
ra_compl_closeSubCatDiff :: (Ord obj, Ord mor) =>
                            RA obj mor -> SubCat obj mor -> SubCatDiff obj mor
ra_compl_closeSubCatDiff c s@(SubCat objs hs) =
  let objects = toListSet $ sub_objects s
      homset a b = toListSet $ lookupDftFM hs zeroSet (a,b)
      compl = ra_compl c
      adds = do a <- objects
                b <- objects
                f <- homset a b
                let g = compl f
                if sub_isMor s a b g then []
                                     else [addToHomset a b g]
  in foldr id (SubCat zeroSet zeroFM) adds
```

For relation algebra closure, we may skip the separate closure operators for division allegories and Dedekind categories:

```
ra_closeStep :: (Ord obj, Ord mor) => RA obj mor -> SubCatClosure obj mor
ra_closeStep c = distrAll_closeStep (ra_distrAll c) `scComp`
                 scStep (ra_compl_closeSubCatDiff c)

subRA s c = sub_ra (scClose (ra_closeStep c) s) c
```

2.3 Matrix Algebra Construction

Concrete relations can usefully be represented as Boolean matrices. We have seen that the Boolean algebra of truth values in itself can already be considered as a relation algebra, the relation algebra of Boolean 1×1-matrices.

We now generalise the construction of matrix relation algebras to coefficients stemming from arbitrary relation algebras, or, for simpler structures, to coefficients from distributive allegories.

2.3.1 Matrix Categories

Given a base allegory or relation algebra, we now want to define matrix algebras over this base. Objects of the matrix algebra are going to be lists of objects of the base, and morphisms are going to be matrices of morphisms of the base, where source and target depend on the position in the matrix.

Composition will be based on an appropriate variant of the skalar product: we have to use composition as multiplication, and join as addition — therefore, already for defining just a category of matrices, we need coefficients from a distributive allegory.

Theorem 2.3.1 If $\mathbf{C} = (Obj_{\mathbf{C}}, Mor_{\mathbf{C}}, \_ : \_ \leftrightarrow \_, \mathbb{I}, ;, \breve{}, \sqcap, \sqcup, \perp\!\!\!\perp)$ is a distributive allegory, then a category $Mat_{\mathbf{C}}$ may be defined as follows:

- objects of $Mat_{\mathbf{C}}$ are finite sequences of objects of $\mathbf{C}$,
- for two objects $\mathcal{A} = [\mathcal{A}_1, \ldots, \mathcal{A}_a]$ and $\mathcal{B} = [\mathcal{B}_1, \ldots, \mathcal{B}_b]$ of $Mat_{\mathbf{C}}$, the associated homset $Hom_{Mat_{\mathbf{C}}}[\mathcal{A}, \mathcal{B}]$ contains all matrices $(f_{i,j})_{i \in \{1,\ldots,a\}, j \in \{1,\ldots,b\}}$ for which for every $i \in \{1, \ldots, a\}$ and every $j \in \{1, \ldots, b\}$ the coefficient $f_{i,j}$ is a homomorphism from $\mathcal{A}_i$ to $\mathcal{B}_j$ in $\mathbf{C}$,
- given three objects $\mathcal{A} = [\mathcal{A}_1, \ldots, \mathcal{A}_a]$, $\mathcal{B} = [\mathcal{B}_1, \ldots, \mathcal{B}_b]$, and $\mathcal{C} = [\mathcal{C}_1, \ldots, \mathcal{C}_c]$, and two morphisms $R : \mathcal{A} \rightarrow \mathcal{B}$, and $S : \mathcal{B} \rightarrow \mathcal{C}$, their composition is defined by the following:
$$(R;S)_{i,k} := \bigsqcup_{j \in \{1,\ldots,b\}} (R_{i,j};S_{j,k}) \qquad \text{for all } i \in \{1, \ldots, a\} \text{ and } j \in \{1, \ldots, c\},$$
- for an object $\mathcal{A} = [\mathcal{A}_1, \ldots, \mathcal{A}_m]$, the identity morphism $\mathbb{I}_{\mathcal{A}} : \mathcal{A} \rightarrow \mathcal{A}$ is defined by
$$(\mathbb{I}_{\mathcal{A}})_{i,j} = \begin{cases} \mathbb{I}_{\mathcal{A}_i} & \text{if } i = j \\ \perp\!\!\!\perp_{\mathcal{A}_i,\mathcal{A}_j} & \text{if } i \neq j \end{cases}$$

Proof: Well-definedness of identity and composition morphisms is obvious. Associativity of composition and the identity properties are shown by standard matrix arguments as follows.

Given four objects

$$\mathcal{A} = [\mathcal{A}_1, \dots, \mathcal{A}_a], \mathcal{B} = [\mathcal{B}_1, \dots, \mathcal{B}_b], \mathcal{C} = [\mathcal{C}_1, \dots, \mathcal{C}_c], \text{ and } \mathcal{D} = [\mathcal{D}_1, \dots, \mathcal{D}_d],$$

and three morphisms $R : \mathcal{A} \to \mathcal{B}$, $S : \mathcal{B} \to \mathcal{C}$, and $T : \mathcal{C} \to \mathcal{D}$, we can prove the associativity of composition by the following calculation:

$$\begin{array}{cll}
 & ((R;S);T)_{i,l} & \\
= & \bigsqcup_k ((R;S)_{i,k};T_{k,l}) & \\
= & \bigsqcup_k ((\bigsqcup_j (R_{i,j};S_{j,k}));T_{k,l}) & \\
= & \bigsqcup_k (\bigsqcup_j ((R_{i,j};S_{j,k});T_{k,l})) & \text{join-distributivity in } \mathbf{C} \\
= & \bigsqcup_k (\bigsqcup_j (R_{i,j};(S_{j,k};T_{k,l}))) & \text{associativity of composition in } \mathbf{C} \\
= & \bigsqcup_j (\bigsqcup_k (R_{i,j};(S_{j,k};T_{k,l}))) & \text{commutativity and associativity of join in } \mathbf{C} \\
= & \bigsqcup_j (R_{i,j};(\bigsqcup_k (S_{j,k};T_{k,l}))) & \text{join-distributivity in } \mathbf{C} \\
= & \bigsqcup_j (R_{i,j};(S;T)_{j,l}) & \\
= & (R;(S;T))_{i,l} &
\end{array}$$

Right-identity:

$$\begin{array}{rcl}
(R;\mathbb{I}_{\mathcal{B}})_{i,j} & = & \bigsqcup_{j' \in \{1,\dots,b\}} (R_{i,j};(\mathbb{I}_{\mathcal{B}})_{j',j}) \\
 & = & R_{i,j};(\mathbb{I}_{\mathcal{B}})_{j,j} \sqcup \bigsqcup_{j' \in \{1,\dots,b\}-\{j\}} (R_{i,j};(\mathbb{I}_{\mathcal{B}})_{j',j}) \\
 & = & R_{i,j};\mathbb{I}_{\mathcal{B}_j} \sqcup \bigsqcup_{j' \in \{1,\dots,b\}-\{j\}} (R_{i,j};\perp\!\!\!\perp_{\mathcal{B}_{j'},\mathcal{B}_j}) \\
 & = & R_{i,j} \sqcup \bigsqcup_{j' \in \{1,\dots,b\}-\{j\}} \perp\!\!\!\perp_{\mathcal{A}_i,\mathcal{B}_j} \\
 & = & R_{i,j} \sqcup \perp\!\!\!\perp_{\mathcal{A}_i,\mathcal{B}_j} \\
 & = & R_{i,j}
\end{array}$$

Left-identity is shown in an analogous way. □

Note that the choice of composition for the multiplication of coefficients occurring in the definition of composition is not arbitrary; it would be misguided to orient oneself at the meet this composition degenerates to in the case of the simple Boolean lattice of truth values. The meet would not even be well-defined since we have $R_{i,j} : \mathcal{A}_i \leftrightarrow \mathcal{B}_j$ and $S_{j,k} : \mathcal{B}_j \leftrightarrow \mathcal{C}_k$.

For the time being, we use a simple list implementation of matrices, but we make it abstract so that we can exchange it later for something more efficient. Therefore we have to provide an explicit export list that makes the names of the abstract types `Vec` and `MatMor` available to importing modules, but hides their implementation:

```
module Matrix(catMat,allMat,distrAllMat,divAllMat,dedMat,raMat
             ,Vec(),vec,unVec,MatMor(),matMor,unMatMor,matMorMap,bM
             ,matMap,matZipWith
             ) where

import RelAlg
import List(nub)
import qualified List(transpose)
```

Objects are just lists of objects of the base category:

```
newtype Vec a = Vec [a] deriving (Eq, Ord, Show, Read)
vec = Vec
unVec (Vec s) = s
```

For the matrix itself we use the usual list-of-lists approach:

```
type Mat a = [[a]]
matMap = map . map
matZipWith = zipWith . zipWith
```

It is important to note that we demand the following **consistency condition**: A matrix representing a morphism $R : [\mathcal{A}_1, \ldots, \mathcal{A}_a] \leftrightarrow [\mathcal{B}_1, \ldots, \mathcal{B}_b]$ is a list with exactly a elements (called *rows*), each of which is a list containing exactly b elements. For ease of implementation we demand this also for cases where a or b are zero.

In order to be able to reconstruct source and target of a morphism even in these cases, we need to include the source and target object lists with the matrix proper in our morphism type:

```
newtype MatMor obj mor = MatMor (Mat mor, [obj], [obj])
                                      deriving (Eq, Ord, Show, Read)
```

We export a variant of the constructor without checking the consistency condition; for this purpose one may use `cat_isMor` from below:

```
matMor m s t = MatMor (m, s, t)
unMatMor (MatMor tr) = tr

matMorMap f (MatMor (m, s, t)) = MatMor (matMap f m, s, t)

bM m = let s = replicate (length m) ()
           t = replicate (length (head m)) ()
       in MatMor (m, s, t)
```

At the heart of the composition of matrices is the "skalar product", which needs to be given $\mathcal{A}_x$ and $\mathcal{C}_y$ along with the row $[R_{x,1}, \ldots, R_{x,b}]$ and the column $[S_{1,y}, \ldots, S_{b,y}]$ because both might be empty and we still need to find the correct bottom element:

```
skalprod :: DistrAll obj mor -> (obj,[mor]) -> (obj,[mor]) -> mor
skalprod da (a,ms1) (b,ms2) =
   foldr (distrAll_join da) (distrAll_bottom da a b) $
   zipWith (distrAll_comp da) ms1 ms2
```

We shall obtain the columns needed for the skalar products by transposition of the matrix. Because of our consistency condition, we need to be careful when transposing empty matrices, where we need non-empty results if the original target object vector is non-empty:

```
transpose :: MatMor obj mor -> MatMor obj mor
transpose (MatMor (m, s, t)) =
  let m' = if null s then map (const []) t
           else List.transpose m
  in MatMor (m', t, s)
```

An identity morphism is easily constructed:

```
matIdmor :: DistrAll obj mor -> [obj] -> Mat mor
matIdmor _   []     = []
matIdmor all (a:as) =
  (distrAll_idmor all a : map (\ a' -> distrAll_bottom all a a') as) :
  zipWith (\ a' ms -> distrAll_bottom all a' a : ms) as (matIdmor all as)
```

For generating the list of all morphisms (in the finite case only) we first use `shape` to generate a matrix containing the respective object pairs:

```
shape :: [a] -> [b] -> Mat (a,b)
shape as bs = [[(a,b)| b <- bs] | a <- as]
```

This matrix is then instantiated in all possible ways by providing the `homset` function as first argument to the following:

```
instantiate :: (a -> [b]) -> Mat a -> [Mat b]
instantiate g m =
  let -- inst' :: [a] -> [[b]]
      inst' [] = [[]]
      inst' (a:as) = [b:bs | b <- g a , bs <- inst' as]
      -- inst'' :: [[a]] -> [Mat b]
      inst'' [] = [[]]
      inst'' (as:ass) = [bs:bss | bs <- inst' as, bss <- inst'' ass]
  in inst'' m
```

That is all we need to define a matrix category:

```
catMat :: (Ord obj, Eq mor) => DistrAll obj mor -> [[obj]]
                            -> Cat (Vec obj) (MatMor obj mor)
catMat da objss = let
    objs = map Vec $ nub objss
 in Cat
  {cat_isObj   = (`elem` objs)
  ,cat_isMor   = (\ (Vec s) (Vec t) (MatMor (mss, s', t')) ->
                    s == s' && t == t' &&
                    length mss == length s' &&
                    let lt = length t' in
                    all (\ row -> length row == lt) mss &&
                    and (do (a,ms) <- zip s mss
                            (b,m) <- zip t ms
                            return $ distrAll_isMor da a b m))
  ,cat_objects = objs
  ,cat_homset  = (\ (Vec s) (Vec t) ->
                     let sh = shape s t
                         mats = instantiate (uncurry $ distrAll_homset da) sh
                     in map (\m -> MatMor (m,s,t)) mats)
  ,cat_source  = (\ (MatMor (_,s,_)) -> Vec s)
  ,cat_target  = (\ (MatMor (_,_,t)) -> Vec t)
  ,cat_idmor   = (\ (Vec s) -> MatMor (matIdmor da s, s, s))
  ,cat_comp    = (\ (MatMor (mss1,s1,t1)) m2@(MatMor (_ ,s2,t2)) ->
                    if t1 /= s2
                    then error ("matrix composition type error " ++
                                show (length t1) ++ ' ' : show (length s2))
                    else let MatMor (mss2T,_,_) = transpose m2
                             mss2C = zip t2 mss2T
                             mkline ms1 = map (skalprod da ms1) mss2C
                             mat = map mkline (zip s1 mss1)
                         in MatMor (mat, s1, t2))
  }
```

2.3.2 Matrix Allegories

With coefficients from a distributive allegory, the additional allegory operations are easily added to a matrix category:

Theorem 2.3.2 If $\mathbf{C} = (Obj_{\mathbf{C}}, Mor_{\mathbf{C}}, \_ : \_ \leftrightarrow \_, \mathbb{I}, ;, \breve{}, \sqcap, \sqcup, \perp\!\!\!\perp)$ is a distributive allegory, then $Mat_{\mathbf{C}}$ may be extended to an allegory by defining, for any two objects $\mathcal{A} = [\mathcal{A}_1, \dots, \mathcal{A}_a]$ and $\mathcal{B} = [\mathcal{B}_1, \dots, \mathcal{B}_b]$,

- the converse of any morphism $R : \mathcal{A} \leftrightarrow \mathcal{B}$ as follows:

$$(R\breve{})_{j,i} = (R_{i,j})\breve{} \qquad \text{for all } i \in \{1, \dots, a\} \text{ and } j \in \{1, \dots, b\},$$

- the meet of any two morphisms $R, S : \mathcal{A} \leftrightarrow \mathcal{B}$ component-wise:

$$(R \sqcap S)_{i,j} = R_{i,j} \sqcap S_{i,j} \qquad \text{for all } i \in \{1, \dots, a\} \text{ and } j \in \{1, \dots, b\};$$

- inclusion between any two morphisms $R, S : \mathcal{A} \leftrightarrow \mathcal{B}$ is then component-wise inclusion:

$$R \sqsubseteq S \quad \Leftrightarrow \quad \forall i \in \{1, \dots, a\}, j \in \{1, \dots, b\} \bullet R_{i,j} \sqsubseteq S_{i,j} \ .$$

Proof: Definition of inclusion from meet, lattice properties of meet, distribution of converse over meet, and that converse is an involution all follow directly from the component-wise definitions.

Still to be checked are the following:

- distribution of converse over composition:

$$\begin{array}{rcl}((R;S)\breve{})_{k,i} & = & ((R;S)_{i,k})\breve{} \\ & = & (\bigsqcup_j (R_{i,j};S_{j,k}))\breve{} \\ & = & \bigsqcup_j (R_{i,j};S_{j,k})\breve{} \\ & = & \bigsqcup_j ((S_{j,k})\breve{};(R_{i,j})\breve{}) \\ & = & \bigsqcup_j ((S\breve{})_{k,j};(R\breve{})_{j,i}) \\ & = & (S\breve{};R\breve{})_{k,i}\end{array}$$

- meet-subdistributivity:

$$\begin{array}{rcl}(Q;(R \sqcap S))_{i,k} & = & \bigsqcup_j (Q_{i,j};(R \sqcap S)_{j,k}) \\ & = & \bigsqcup_j (Q_{i,j};(R_{j,k} \sqcap S_{j,k})) \\ & \sqsubseteq & \bigsqcup_j (Q_{i,j};R_{j,k} \sqcap Q_{i,j};S_{j,k}) \\ & \sqsubseteq & (\bigsqcup_j (Q_{i,j};R_{j,k})) \sqcap \bigsqcup_l (Q_{i,l};S_{l,k}) \\ & = & (Q;R)_{i,k} \sqcap (Q;S)_{i,k} \\ & = & (Q;R \sqcap Q;S)_{i,k}\end{array}$$

- modal rule:

$$\begin{array}{rcl}(Q;R \sqcap S)_{i,k} & = & (Q;R)_{i,k} \sqcap S_{i,k} \\ & = & (\bigsqcup_j Q_{i,j};R_{j,k}) \sqcap S_{i,k} \\ & = & \bigsqcup_j (Q_{i,j};R_{j,k} \sqcap S_{i,k}) \\ & \sqsubseteq & \bigsqcup_j ((Q_{i,j} \sqcap S_{i,k};(R_{j,k})\breve{});R_{j,k}) \\ & = & \bigsqcup_j ((Q_{i,j} \sqcap S_{i,k};(R\breve{})_{k,j});R_{j,k}) \\ & \sqsubseteq & \bigsqcup_j ((Q_{i,j} \sqcap \bigsqcup_{k'} (S_{i,k'};(R\breve{})_{k',j}));R_{j,k}) \\ & = & \bigsqcup_j ((Q_{i,j} \sqcap (S;R\breve{})_{i,j});R_{j,k}) \\ & = & \bigsqcup_j ((Q \sqcap S;R\breve{})_{i,j}));R_{j,k}) \\ & = & ((Q \sqcap S;R\breve{});R)_{i,k}\end{array}$$

□

For implementing the matrix allegory, we therefore need component-wise definitions for meet and inclusion, and for conversion we not only have to transpose the matrix (carefully, see above), but also converse every coefficient:

```
allMat :: (Ord obj, Eq mor) => DistrAll obj mor -> [[obj]]
                            -> All (Vec obj) (MatMor obj mor)
allMat da objss = All
  {all_cat = catMat da objss
  ,all_converse = (\ m1 -> let MatMor (m, s, t) = transpose m1
                               m' = matMap (distrAll_converse da) m
                           in MatMor (m', s, t))
  ,all_meet = (\ (MatMor (mss1,s1,t1)) (MatMor (mss2,s2,t2)) ->
                 if s1 /= s2
                 then error ("matrix meet source type error")
                 else if t1 /= t2
                 then error ("matrix meet target type error")
                 else let mat = zipWith (zipWith (distrAll_meet da)) mss1 mss2
                      in MatMor (mat,s1,t1))
  ,all_incl = (\ (MatMor (mss1,s1,t1)) (MatMor (mss2,s2,t2)) ->
                 if s1 /= s2
                 then error ("matrix inclusion source type error")
                 else if t1 /= t2
                 then error ("matrix inclusion target type error")
                 else all and $ zipWith (zipWith (distrAll_incl da)) mss1 mss2)
  }
```

2.3.3 Distributive Allegories

The component-wise definitions of the additional components bottom and join make most of the required laws trivial:

Theorem 2.3.3 If $\mathbf{C} = (Obj_{\mathbf{C}}, Mor_{\mathbf{C}}, \_ : \_ \leftrightarrow \_, \mathbb{I}, ;, \breve{}, \sqcap, \sqcup, \perp\!\!\!\perp)$ is a distributive allegory, then $Mat_{\mathbf{C}}$ may be extended to a distributive allegory by defining, for any two objects $\mathcal{A} = [\mathcal{A}_1, \ldots, \mathcal{A}_a]$ and $\mathcal{B} = [\mathcal{B}_1, \ldots, \mathcal{B}_b]$,

- the zero morphism $\perp\!\!\!\perp_{\mathcal{A},\mathcal{B}} : \mathcal{A} \leftrightarrow \mathcal{B}$ as follows:

$$(\perp\!\!\!\perp_{\mathcal{A},\mathcal{B}})_{i,j} = \perp\!\!\!\perp_{\mathcal{A}_i,\mathcal{B}_j} \qquad \text{for all } i \in \{1,\ldots,a\} \text{ and } j \in \{1,\ldots,b\},$$

- the join of any two morphisms $R, S : \mathcal{A} \leftrightarrow \mathcal{B}$ component-wise:

$$(R \sqcup S)_{i,j} = R_{i,j} \sqcup S_{i,j} \qquad \text{for all } i \in \{1,\ldots,a\} \text{ and } j \in \{1,\ldots,b\}.$$

Proof: The lattice properties of join and the zero law are trivial; we only show join-distributivity:

$$\begin{aligned}
(Q;(R \sqcup S))_{i,k} &= \textstyle\bigsqcup_j (Q_{i,j};(R \sqcup S)_{j,k}) \\
&= \textstyle\bigsqcup_j (Q_{i,j};(R_{j,k} \sqcup S_{j,k})) \\
&= \textstyle\bigsqcup_j (Q_{i,j};R_{j,k} \sqcup Q_{i,j};S_{j,k}) \\
&= (\textstyle\bigsqcup_j (Q_{i,j};R_{j,k})) \sqcup \textstyle\bigsqcup_j (Q_{i,j};S_{j,k}) \\
&= (Q;R)_{i,k} \sqcup (Q;S)_{i,k} \\
&= (Q;R \sqcup Q;S)_{i,k}
\end{aligned}$$

□

Accordingly, defining bottom is easy:

```
bottomMat da as bs = [bottomRow da a bs | a <- as]

bottomRow da a bs = map (distrAll_bottom da a) bs
```

To obtain a list of atoms — we treat the global atom list and the list of atoms contained in a given morphism in parallel — is, however, slightly more effort:

```
atomMats da [] bs = []
atomMats da [a] bs = map (:[]) (atomRows da a bs)
atomMats da (a:as) bs =
  map (: (bottomMat da as bs)) (atomRows da a bs) ++
  map ((bottomRow da a bs) :) (atomMats da as bs)

atomsMats da [] bs _ = []
atomsMats da [a] bs [r] = map (:[]) (atomsRows da a bs r)
atomsMats da (a:as) bs (r:rs) =
  map (: (bottomMat da as bs)) (atomsRows da a bs r) ++
  map ((bottomRow da a bs) :) (atomsMats da as bs rs)
atomsMats _ _ _ _ = error "atomsMats"

atomRows da a []     = [[]]
atomRows da a [b] = map (:[]) (distrAll_atomset da a b)
atomRows da a (b:bs) =
  map (: (bottomRow da a bs)) (distrAll_atomset da a b) ++
  map ((distrAll_bottom da a b) :) (atomRows da a bs)

atomsRows da a [] _    = [[]]
atomsRows da a [b] [m] = map (:[]) (distrAll_atoms da m)
atomsRows da a (b:bs) (m:ms) =
  map (: (bottomRow da a bs)) (distrAll_atoms da m) ++
  map ((distrAll_bottom da a b) :) (atomsRows da a bs ms)
atomsRows _ _ _ _ = error "atomsRows"
```

These definitions are not adequate if empty objects are involved, so we have to treat these cases separately:

```
distrAllMat :: (Ord obj, Eq mor) => DistrAll obj mor -> [[obj]]
                              -> DistrAll (Vec obj) (MatMor obj mor)
distrAllMat da objss = DistrAll
  {distrAll_all  = allMat da objss
  ,distrAll_bottom = (\ (Vec s) (Vec t) ->
                    let mat = bottomMat da s t
                    in MatMor (mat, s, t))
  ,distrAll_join = (\ (MatMor (mss1,s1,t1)) (MatMor (mss2,s2,t2)) ->
                  if s1 /= s2
                  then error ("matrix join source type error")
                  else if t1 /= t2
                  then error ("matrix join target type error")
                  else let mat = zipWith (zipWith (distrAll_join da)) mss1 mss2
                       in MatMor (mat,s1,t1))
  ,distrAll_atomset = (\ (Vec s) (Vec t) ->
                        if null s || null t then [] else
```

```
                      map (\ m -> MatMor (m,s,t)) $ atomMats da s t)
  ,distrAll_atoms = (\ (MatMor (m, s, t)) ->
                    if null s || null t then [] else
                    map (\n -> MatMor (n,s,t)) $ atomsMats da s t m)
  }
```

2.3.4 Division Allegories

As we shall see, the definition of the residual coefficients is dual to the definition of the composition coefficients. It therefore relies on meet, and, for empty intermediate objects, also on the presence of top as the unit of meet.

Since we do not want to differentiate between matrix algebras with and without empty objects, we therefore need coefficients from a Dedekind category:

Theorem 2.3.4 If $\mathbf{C} = (Obj_{\mathbf{C}}, Mor_{\mathbf{C}}, \_ : \_ \leftrightarrow \_, \mathbb{I}, ;, \breve{}, \sqcap, \sqcup, \perp\!\!\!\perp, \backslash, /, \mathbb{T})$ is a Dedekind category, then $Mat_{\mathbf{C}}$ may be extended to a division allegory by defining, for any three objects $\mathcal{A} = [\mathcal{A}_1, \dots, \mathcal{A}_a]$, $\mathcal{B} = [\mathcal{B}_1, \dots, \mathcal{B}_b]$, and $\mathcal{C} = [\mathcal{C}_1, \dots, \mathcal{C}_c]$, and any three matrix morphisms $Q : \mathcal{A} \leftrightarrow \mathcal{C}$, $R : \mathcal{A} \leftrightarrow \mathcal{B}$, and $S : \mathcal{B} \leftrightarrow \mathcal{C}$, the coefficients of their residuals in terms of the residuals of their coefficients:

$$\begin{array}{rcl} (R\backslash Q)_{j,k} & := & \bigsqcap_i (R_{i,j}\backslash Q_{i,k}) \\ (Q/S)_{i,j} & := & \bigsqcap_k (Q_{i,k}/S_{j,k}) \end{array}$$

Proof: We only carry out the proof for the left residual:

$$\begin{array}{rcl} R \sqsubseteq (Q/S) & \Leftrightarrow & \forall i,j \bullet R_{i,j} \sqsubseteq (Q/S)_{i,j} \\ & \Leftrightarrow & \forall i,j \bullet R_{i,j} \sqsubseteq \bigsqcap_k (Q_{i,k}/S_{j,k}) \\ & \Leftrightarrow & \forall i,j \bullet \forall k \bullet R_{i,j} \sqsubseteq Q_{i,k}/S_{j,k} \\ & \Leftrightarrow & \forall i,j,k \bullet R_{i,j};S_{j,k} \sqsubseteq Q_{i,k} \\ & \Leftrightarrow & \forall i,k \bullet \forall j \bullet R_{i,j};S_{j,k} \sqsubseteq Q_{i,k} \\ & \Leftrightarrow & \forall i,k \bullet (\bigsqcup_j (R_{i,j};S_{j,k})) \sqsubseteq Q_{i,k} \\ & \Leftrightarrow & \forall i,k \bullet (R;S)_{i,k} \sqsubseteq Q_{i,k} \\ & \Leftrightarrow & R;S \sqsubseteq Q \end{array}$$ □

We first define the common structure of both residuals in an auxiliary function:

```
skalres :: Ded obj mor -> (mor -> mor -> mor) ->
                          (obj,[mor]) -> (obj,[mor]) -> mor
skalres d res (a,ms1) (b,ms2) =
   foldr (ded_meet d) (ded_top d a b) $
   zipWith res ms1 ms2
```

The right residual now needs two transpositions to get the columns lined up properly, while the left residual directly uses the rows. For the symmetric quotient, we simply use the default definition:

```
divAllMat :: (Ord obj, Eq mor) => Ded obj mor -> [[obj]]
                            -> DivAll (Vec obj) (MatMor obj mor)
divAllMat d objss = diva where
 diva = DivAll
  {divAll_distrAll = distrAllMat (ded_distrAll d) objss
  ,divAll_rres = (\ (m1@(MatMor (_,s1,t1))) m2@(MatMor (_ ,s2,t2)) ->
                   if s1 /= s2
                   then error ("matrix right residual type error " ++
                               show (length s1) ++ ' ' : show (length s2))
                   else let MatMor (mss1T,_,_) = transpose m1
                            MatMor (mss2T,_,_) = transpose m2
                            mss1C = zip t1 mss1T
                            mss2C = zip t2 mss2T
                            mkline ms1 = map (skalres d (ded_rres d) ms1) mss2C
                            mat = map mkline (zip t1 mss1T)
                        in MatMor (mat, t1, t2))
  ,divAll_lres = (\ (MatMor (mss1,s1,t1)) (MatMor (mss2,s2,t2)) ->
                   if t1 /= t2
                   then error ("matrix left residual type error " ++
                               show (length t1) ++ ' ' : show (length t2))
                   else let mss2C = zip s2 mss2
                            mkline ms1 = map (skalres d (ded_lres d) ms1) mss2C
                            mat = map mkline (zip s1 mss1)
                        in MatMor (mat, s1, s2))
  ,divAll_syq  = divAll_syq_default diva
  }
```

2.3.5 Dedekind Categories and Relation Algebras

Since we already have a Dedekind category at the coefficient level, getting the top morphism is now easy again, and complement at the coefficient level is lifted component-wise to yield matrix complements:

Theorem 2.3.5 If $\mathbf{C} = (Obj_{\mathbf{C}}, Mor_{\mathbf{C}}, \_ : \_ \leftrightarrow \_, \mathbb{I}, ;, \breve{}, \sqcap, \sqcup, \perp\!\!\!\perp, \backslash, /, \top\!\!\!\top)$ is a Dedekind category, then $Mat_{\mathbf{C}}$ may be extended to a Dedekind category by defining, for any two objects $\mathcal{A} = [\mathcal{A}_1, \dots, \mathcal{A}_a]$ and $\mathcal{B} = [\mathcal{B}_1, \dots, \mathcal{B}_b]$, the top morphism component-wise as follows:

$$(\top\!\!\!\top_{\mathcal{A},\mathcal{B}})_{i,j} = \top\!\!\!\top_{\mathcal{A}_i,\mathcal{B}_j}$$

Also, if $\mathbf{C}$ is a relation algebra, then $Mat_{\mathbf{C}}$ may be extended to a relation algebra by defining, for any two objects $\mathcal{A} = [\mathcal{A}_1, \dots, \mathcal{A}_a]$ and $\mathcal{B} = [\mathcal{B}_1, \dots, \mathcal{B}_b]$, and for any matrix morphism $R : \mathcal{A} \leftrightarrow \mathcal{B}$, the complement component-wise as follows:

$$(\overline{R})_{i,j} = \overline{R_{i,j}}$$

Proof: All remaining properties follow by simple component-wise reasoning. A direct proof of the Schröder rule is the following:

$$
\begin{array}{rcll}
R;S \sqsubseteq Q & \Leftrightarrow & \forall i,k \bullet (R;S)_{i,k} \sqsubseteq Q_{i,k} & \\
 & \Leftrightarrow & \forall i,k \bullet \bigsqcup_j (R_{i,j};S_{j,k}) \sqsubseteq Q_{i,k} & \\
 & \Leftrightarrow & \forall i,k,j \bullet R_{i,j};S_{j,k} \sqsubseteq Q_{i,k} & \\
 & \Leftrightarrow & \forall i,k,j \bullet R\breve{}_{j,i};\overline{Q}_{i,k} \sqsubseteq \overline{S}_{j,k} & \text{Schröder for coefficients} \\
 & \Leftrightarrow & \forall k,j \bullet \bigsqcup_i (R\breve{}_{j,i};\overline{Q}_{i,k}) \sqsubseteq \overline{S}_{j,k} & \\
 & \Leftrightarrow & \forall k,j \bullet (R\breve{};\overline{Q})_{j,k} \sqsubseteq \overline{S}_{j,k} & \\
 & \Leftrightarrow & R\breve{};\overline{Q} \sqsubseteq \overline{S} & \square
\end{array}
$$

The remaining Haskell definitions are therefore completely straightforward:

```
dedMat :: (Ord obj, Eq mor) => Ded obj mor -> [[obj]]
                            -> Ded (Vec obj) (MatMor obj mor)
dedMat ded objss = Ded
  {ded_divAll = divAllMat ded objss
  ,ded_top  = (\ (Vec s) (Vec t) ->
                  let mat = [[ded_top ded a b | b <- t] | a <- s]
                  in MatMor (mat, s, t))
  }

raMat ra objss = RA
  {ra_ded   = dedMat (ra_ded ra) objss
  ,ra_compl = (\ (MatMor (m, s, t)) ->
                 MatMor (matMap (ra_compl ra) m, s, t))
  }
```

2.4 Construction Based on Atom Sets

According to the definition, every homset of a relation algebra is an atomic Boolean lattice, and the structure of atomic Boolean lattices is completely determined by the set of atoms. Together with join-distributivity and isotonicity of converse, every relation algebra is therefore completely determined by the atom sets of its homsets, and by the behaviour of converse and composition on these atoms.

We now use this fact to arrive at a more economic way of defining relation algebras.

For an example that comes with a detailed explanation of this principle see Sect. 3.1.

Since we keep the morphism data type of atom set categories abstract, we have to provide an explicit export list for this module:

```
module Atomset(ACat(..),acat_idmor_default,acat_idmor_defaultM
              ,SetMor(),mkSetMor,unSetMor,atmor
              ,atomsetCat,acat_TEST,acat_TEST'
              ,AAll(..),aall_isObj,aall_isAtom,aall_objects
                       ,aall_atomset,aall_idmor,aall_comp
              ,atomsetAll,atomsetDistrAll,atomsetDivAll,atomsetDed,atomsetRA
              ,aall_TEST
              ,showsAtomset0,showsAtomset', showsAtomset
              ,showsAtCompEntry0,showsAtCompDefault,showsAtCompEntry1
              ,showsAtComp0,showsAtComp', showsAtComp
              ,showsIdmor0,showsIdmor', showsIdmor
              ,showsACat0,showsACat', showsACat
              ,showsConv0,showsAtConv', showsAtConv
              ,showsAAll0,showsAAll', showsAAll
              ,showsARA0,showsARA', showsARA
              ,boolMatARASchows,writeBoolMatARA
              ,Cycle,cycleRepresentatives, cycles
              ,AtomCompTable,addCycle,tableAtComp,negTableAtComp
              ,allCycles,showsCycAtComp
              ,acatB,aallB
              ,distrAll_acat,distrAll_aall
              ,divAll_acat,divAll_aall
              ,ded_acat,ded_aall
              ,ra_acat,ra_aall
              ,MatAt,acatMat,aallMat
              ,matBtoAtCat,atCatToMatB
              ) where

import RelAlg
import Matrix

import FiniteMaps
import Sets

import List (nub,sort)
import ExtPrel
```

2.4.1 Atom Category Definitions

If we intend the morphisms of a category to be sets of elements of some base set (and we call these elements “atoms” for their intended rôle in relation algebras), if identical atoms are to be allowed to occur in different homsets, and if composition should preserve joins ad meets over these sets, then such a category is determined by the following items:

- its objects,
- for any two objects, the atoms of the respective homset, and
- for any three objects and two atoms (from the respective homsets), the set of atoms that occur in the composition of the two atoms.

As for full categories, we complete this list with well-definedness predicates for objects and atoms, and with information about the identity morphisms:

```
data ACat obj atom = ACat
  {acat_isObj    :: obj -> Bool
  ,acat_isAtom   :: obj -> obj -> atom -> Bool
  ,acat_objects  :: [obj]
  ,acat_atomset  :: obj -> obj -> [atom]
  ,acat_idmor    :: obj -> [atom]
  ,acat_comp     :: obj -> obj -> obj -> atom -> atom -> [atom]
  }
```

If such an atom category definition is well-defined, then it is redundant; in particular the information about identity atoms can be derived from the enumerations and composition:

```
acat_idmor_default :: Eq atom => ACat obj atom -> obj -> [atom]
acat_idmor_default ac o =
  let as = acat_atomset ac o o
      os = acat_objects ac
      testL p a b = all (`elem` [b]) (acat_comp ac o o p a b)
      testO p a = and (map (testL p a) (acat_atomset ac o p))
      reduceO as p = filter (testO p) as
  in foldl reduceO as os
```

Whenever we want to actually use this default when defining an atom category description, we can considerably speed up access to the identity by memorising it; since demanding `Ord` for objects is not a heavy constraint, and the overhead for finite maps of the sizes we shall usually need will be neglegible, we use finite maps instead of arrays for memoisation, and, as usual, have to provide the domain for memoisation explicitly:

```
acat_idmor_defaultFM :: (Ord obj, Eq atom) =>
                       ACat obj atom -> FiniteMap obj x -> obj -> [atom]
acat_idmor_defaultFM ac dom = memoFMfm' dom (acat_idmor_default ac)
```

As an abbreviation, we use the whole object list as the domain, with the "`M`" standing for memoisation:

```
acat_idmor_defaultM :: (Ord obj, Eq atom) => ACat obj atom -> obj -> [atom]
acat_idmor_defaultM ac =
   let dom = listToFM $ zip (acat_objects ac) (repeat ())
   in acat_idmor_defaultFM ac dom
```

2.4.2 Building Categories from Atom Category Definitions

We introduce an abstract data type for morphisms built from sets of atoms.

Since in categories, we need to be able to identify source and target of a morphism, we have to explicitly include that information here (even if it was included in atoms, we still would need it for the empty set).

```
newtype SetMor obj mor = SetMor (Set mor,obj,obj) deriving (Show, Read)

unSetMor (SetMor t) = t
unSetMor' (SetMor (ms,s,t)) = (toListSet ms, s, t)

mkSetMor a b as = SetMor (listToSet as, a, b)
```

We have more tools available when morphisms are in `Eq` and `Ord`; since these instances are not included in the set package we use, we rely on the (undocumented) feature that `toListSet` always returns an ordered list of unique elements:

```
instance (Eq obj, Ord mor) => Eq (SetMor obj mor) where
  SetMor (as1,s1,t1) == SetMor (as2,s2,t2) =
   s1 == s2 && t1 == t2 && toListSet as1 == toListSet as2
instance (Ord obj, Ord mor) => Ord (SetMor obj mor) where
  SetMor (as1,s1,t1) <= SetMor (as2,s2,t2) =
    (toListSet as1,s1,t1) <= (toListSet as2,s2,t2)
```

Defining the category is now quite straightforward:

```
atomsetCat :: (Eq obj, Ord mor) => ACat obj mor -> Cat obj (SetMor obj mor)
atomsetCat ac = Cat
  {cat_isObj   = acat_isObj ac
  ,cat_isMor   = (\ s t (SetMor (as,s',t')) ->
                    s == s' && t == t' &&
                    foldSet (\ m b -> acat_isAtom ac s t m && b) True as)
  ,cat_objects = acat_objects ac
  ,cat_homset  = (\ a b -> let atoms = acat_atomset ac a b
                           in map (mkSetMor a b) (power atoms))
  ,cat_source  = (\ (SetMor (as,s,t)) -> s)
  ,cat_target  = (\ (SetMor (as,s,t)) -> t)
  ,cat_idmor   = (\ a -> mkSetMor a a $ acat_idmor ac a)
  ,cat_comp    = (\ (SetMor (as1,s1,t1)) (SetMor (as2,s2,t2)) ->
     if t1 /= s2 then error "atomsetCat.comp type error" else
     SetMor (foldSet (\ a1 s ->
                         foldSet (\ a2 s ->
                             foldr addToSet s (acat_comp ac s1 s2 t2 a1 a2)
                          ) s as2
                      ) zeroSet as1
            ,s1,t2))
  }
```

The auxiliary function `power` used to generate homsets again uses function composition instead of list concatenation for efficiency and may be found in Sect. A.3.

2.4.3 Atom Category Definition Testing

The above definition of `atomsetCat` shows how we can directly test well-definedness of atom category definitions; we group the tests in the following way:

i) One object: Consistency of object list, and of identity as atom set

ii) Two objects:

   (a) Two objects, one atom: Consistency of atom sets, left-identity

   (b) Two objects, one atom in the other direction: Right identity

iii) Three objects, two atoms: Check whether composition yields consistent atom set

iv) Four objects, three atoms: Associativity of composition

```
acat_TEST :: (Eq obj, Ord atom)  => Test ACat obj atom
acat_TEST c =
  let isObj   = acat_isObj   c
      isAtom  = acat_isAtom  c
      objects = acat_objects c
      atomset = acat_atomset c
      idmor   = acat_idmor   c
      comp    = acat_comp    c
  in
  ffold (do o1 <- objects
            testX (isObj o1) [o1] [] "object list contains non-object"
             (do let i1 = idmor o1
                     test (all (isAtom o1 o1) i1) [o1] i1
                          "identity contains non-atoms" : do
                   o2 <- objects
                   (do f <- atomset o1 o2
                       testX (isAtom o1 o2 f) [o1,o2] [f]
                             "atomset contains non-atom"
                        (let f' = nub $ concat $ do i1a <- i1
                                                    return $ comp o1 o1 o2 i1a f
                         in [test ([f] == f') [o1,o2] (i1++f:f')
                                 "left-identity violated"]
                        )
                   ) ++
                    (do g <- atomset o2 o1
                        let g' = nub $ concat $ do i1a <- i1
                                                   return $ comp o2 o1 o1 g i1a
                        [test ([g] == g') [o2,o1] (i1 ++ g:g')
                              "right-identity violated"]
                    )
             )
        ) .
  ffold (do o1 <- objects
            o2 <- objects
            f <- atomset o1 o2
            o3 <- objects
            g <- atomset o2 o3
            let fg = comp o1 o2 o3 f g
            testX (all (isAtom o1 o3) fg) [o1,o2,o3] (f:g:fg)
                  "composition yields non-atom"
             (do o4 <- objects
                 let os = [o1,o2,o3,o4]
                 h <- atomset o3 o4
                 let gh = comp o2 o3 o4 g h
                 let k1 = sort $ nub (gh >>= comp o1 o2 o4 f)
                 let k2 = sort $ nub (fg >>= flip (comp o1 o3 o4) h)
                 [test (k1 == k2) os [f,g,h] "composition is not associative"]
             )
        )
```

Modularising the test for better readability incurs a runtime cost of about two percent:

```
acat_TEST' :: (Eq obj, Ord atom)  => Test ACat obj atom
acat_TEST' c =
  let isObj   = acat_isObj   c
      isAtom  = acat_isAtom  c
      objects = acat_objects c
      atomset = acat_atomset c
      idmor   = acat_idmor   c
      comp    = acat_comp    c
  in
  ffold (do o1 <- objects
            [test (isObj o1) [o1] [] "object list contains non-object"]
        ) .
  ffold (do o1 <- objects
            let i1 = idmor o1
            [test (all (isAtom o1 o1) i1) [o1] i1 "identity contains non-atoms"]
        ) .
  ffold (do o1 <- objects
            let i1 = idmor o1
            o2 <- objects
            f <- atomset o1 o2
            [test (isAtom o1 o2 f) [o1,o2] [f] "atomset contains non-atom"]
        ) .
  ffold (do o1 <- objects
            let i1 = idmor o1
            o2 <- objects
            let os = [o1,o2]
            let i2 = idmor o2
            f <- atomset o1 o2
```

```
          let f'  = nub $ concat $ do i1a <- i1
                                      return $ comp o1 o1 o2 i1a f
          let f'' = nub $ concat $ do i2a <- i2
                                      return $ comp o1 o2 o2 f i2a
          [test ([f] == f' ) os (i1 ++ f:f' ) "left-identity violated" .
           test ([f] == f'') os (i2 ++ f:f'') "right-identity violated"]
      ) .
  ffold (do o1 <- objects
          o2 <- objects
          f <- atomset o1 o2
          o3 <- objects
          g <- atomset o2 o3
          let fg = comp o1 o2 o3 f g
          [test (all (isAtom o1 o3) fg) [o1,o2,o3] (f:g:fg)
                "composition yields non-atom"]
      ) .
  ffold (do o1 <- objects
          o2 <- objects
          f <- atomset o1 o2
          o3 <- objects
          g <- atomset o2 o3
          let fg = comp o1 o2 o3 f g
          o4 <- objects
          let os = [o1,o2,o3,o4]
          h <- atomset o3 o4
          let gh = comp o2 o3 o4 g h
          let k1 = sort $ nub (gh >>= comp o1 o2 o4 f)
          let k2 = sort $ nub (fg >>= flip (comp o1 o3 o4) h)
          [test (k1 == k2) os [f,g,h] "composition is not associative"]
       )
```

2.4.4 From Allegories to Relation Algebras

With the atom set category definitions from above, we already have homsets that are atomic complete Boolean lattices, and (sub-)distributivity of composition over join and meet. However, we do not yet have even an allegory, because information about converse is still missing — note that the converse of an atom has to be an atom again because of monotony of converse:

```
data AAll obj atom = AAll
  {aall_acat :: ACat obj atom
  ,aall_converse :: obj -> obj -> atom -> atom
  }
```

We expand the interface to comprise that of the included atom category definition:

```
aall_isObj   = acat_isObj   . aall_acat  -- :: obj -> Bool
```

```
aall_isAtom   = acat_isAtom   . aall_acat  -- :: obj -> atom -> Bool
aall_objects  = acat_objects  . aall_acat  -- :: [obj]
aall_atomset  = acat_atomset  . aall_acat  -- :: obj -> obj -> [atom]
aall_idmor    = acat_idmor    . aall_acat  -- :: obj -> [atom]
aall_comp     = acat_comp     . aall_acat  -- :: o -> o -> o -> at -> at -> [at]
```

An allegory is easily constructed:

```
atomsetAll :: (Eq obj, Ord mor) => AAll obj mor -> All obj (SetMor obj mor)
atomsetAll aa = let ac = aall_acat aa
 in All
  {all_cat = atomsetCat ac
  ,all_converse = (\ (SetMor (as,s,t)) ->
                    SetMor (foldSet (addToSet . aall_converse aa s t) zeroSet as
                           ,t,s))
  ,all_meet = (\ (SetMor (as1,s1,t1)) (SetMor (as2,s2,t2)) ->
                if s1 /= s2 then error "atomsetAll.meet source type error" else
                if t1 /= t2 then error "atomsetAll.meet target type error" else
                SetMor (intersectSet as1 as2, s1, t1))
  ,all_incl = (\ (SetMor (as1,s1,t1)) (SetMor (as2,s2,t2)) ->
                if s1 /= s2 then error "atomsetAll.incl source type error" else
                if t1 /= t2 then error "atomsetAll.incl target type error" else
                isZeroSet (diffSet as1 as2))
  }
```

Also for distributive allegories everything is straightforward:

```
atomsetDistrAll :: (Eq obj, Ord mor) => AAll obj mor ->
                                        DistrAll obj (SetMor obj mor)
atomsetDistrAll aa = DistrAll
  {distrAll_all = atomsetAll aa
  ,distrAll_bottom = (\ a b -> SetMor (zeroSet, a, b))
  ,distrAll_join = (\ (SetMor (as1,s1,t1)) (SetMor (as2,s2,t2)) ->
           if s1 /= s2 then error "atomsetDistrAll.join source type error" else
           if t1 /= t2 then error "atomsetDistrAll.join target type error" else
           SetMor (joinSet as1 as2, s1, t1))
  ,distrAll_atomset = (\ a b -> map (atmor a b) $ aall_atomset aa a b)
  ,distrAll_atoms = (\ (SetMor (as,a,b)) -> map (atmor a b) $ toListSet as)
  }

atmor a b at = SetMor (unitSet at, a, b)
```

For division allegories we use a little trick: We know that we already have a relation algebra, so we use the default residual definitions of that relation algebra for division allegories, although, at least formally, that relation algebra is defined in terms of this division allegory. Since there is however no harmful cyclic dependency between the record components involved, everything is well-defined and we do not drop into a “black hole”:

```
atomsetDivAll :: (Eq obj, Ord mor) => AAll obj mor -> DivAll obj (SetMor obj mor)
atomsetDivAll aa = da where
 da = DivAll
  {divAll_distrAll = atomsetDistrAll aa
  ,divAll_rres = ra_rres_default ra
  ,divAll_lres = ra_lres_default ra
  ,divAll_syq  = divAll_syq_default da
  }
 ra = atomsetRA aa
```

For Dedekind categories and relation algebras there are no further problems:

```
atomsetDed :: (Eq obj, Ord mor) => AAll obj mor -> Ded obj (SetMor obj mor)
atomsetDed aa = Ded
  {ded_divAll = atomsetDivAll aa
  ,ded_top    = (\ a b -> mkSetMor a b (aall_atomset aa a b))
  }

atomsetRA :: (Eq obj, Ord mor) => AAll obj mor -> RA obj (SetMor obj mor)
atomsetRA aa = RA
  {ra_ded   = atomsetDed aa
  ,ra_compl = (\ (SetMor (as,s,t)) ->
                 SetMor (listToSet (filter (\ a -> not (a ‘elemSet‘ as))
                                           (aall_atomset aa s t))
                        , s, t))
  }
```

2.4.5 Atom Allegory Definition for $\mathbb{B}$

Just for testing, we provide the second simplest atom allegory definition that is possible:

```
acatB :: ACat () ()
acatB = ACat
  {acat_isObj   = const True
  ,acat_isAtom  = (\ s t a -> True)
  ,acat_objects = [()]
  ,acat_atomset = const $ const [()]
  ,acat_idmor   = const [()]
  ,acat_comp    = (\ a b c f g -> [()])
  }

aallB :: AAll () ()
aallB = AAll
  {aall_acat = acatB
  ,aall_converse = const $ const id
  }
```

With this definition, `atomsetRA aallB` is isomorphic to `raB`; we leave the definition of the functors as an exercise to the reader (see also 2.4.7 and 2.4.11)

2.4.6 Atom Allegory Definition Testing

It is easy to check that, in distributive allegories, the Dedekind formula for $P : \mathcal{A} \leftrightarrow \mathcal{C}$, $Q : \mathcal{A} \leftrightarrow \mathcal{B}$, and $R : \mathcal{B} \leftrightarrow \mathcal{C}$:

$$P \sqcap Q{;}R \sqsubseteq (Q \sqcap P{;}R\breve{}){;}(R \sqcap Q\breve{}{;}P)$$

follows from any of the following:

- $P = P_1 \sqcup P_2$ and the Dedekind formulae for P_1, Q, R and for P_2, Q, R hold, or
- $Q = Q_1 \sqcup Q_2$ and the Dedekind formulae for P, Q_1, R and for P, Q_2, R hold, or
- $R = R_1 \sqcup R_2$ and the Dedekind formulae for P, Q, R_1 and for P, Q, R_2 hold.

Therefore it is sufficient to check the Dedekind formula for all atoms, and we organise the full atom allegory definition test as follows:

i) One object: Preservation of identities by converse

ii) Two objects, one atom: Consistency of result atom of converse, involution test

iii) Three objects, two atoms: Preservation of composition by converse

iv) Three objects, three atoms: Dedekind rule

```
aall_TEST :: (Eq obj, Ord atom)  => Test AAll obj atom
aall_TEST c =
  let isAtom  = aall_isAtom   c
      objects = aall_objects  c
      atomset = aall_atomset  c
      idmor   = aall_idmor    c
      comp    = aall_comp     c
      conv    = aall_converse c
      ameet a l = if a `elem` l then [a] else []
  in
  ffold $ do
    o1 <- objects
    let i1 = idmor o1
    test (all (\ i -> conv o1 o1 i == i) i1) [o1] i1
         "converse does not preserve identity" : do
     o2 <- objects
     q <- atomset o1 o2
     let qC = conv o1 o2 q
     let qCC = conv o2 o1 qC
     (testX (isAtom o2 o1 qC) [o1,o2] [q,qC] "converse yields non-atom" .
      testX (qCC == q) [o1,o2] [q,qC,qCC] "converse not involutory"
      )
      (do
      o3 <- objects
      let os = [o1,o2,o3]
      r <- atomset o2 o3
      let rC = conv o2 o3 r
      let qrC = sort $ nub $ map (conv o1 o3) (comp o1 o2 o3 q r)
      let rCqC = sort $ nub $ comp o3 o2 o1 rC qC
      test (qrC == rCqC) os (q : r : qrC ++ rCqC) "non-functorial converse" : do
        p <- atomset o1 o3
        let p' = p `ameet` (comp o1 o2 o3 q  r )
        let q' = q `ameet` (comp o1 o3 o2 p  rC)
        let r' = r `ameet` (comp o2 o1 o3 qC p )
        let qr' = do qa <- q'; ra <- r'; comp o1 o2 o3 qa ra
        [test (all (`elem` qr') p') os ([p,q,r] ++ qr') "Dedekind violation"]
    )
```

2.4.7 Atom Allegory Definition Output

Once we constructed a relation algebra, we may want to output its definition in a directly reusable form. As an example, consider algebras like `raMat raB [[],[()],[(),()]]`, which (essentially) have the given lists of unit values as objects and small matrices of Booleans as morphisms. We might want to generate dedicated object and atom data types like

```
-- data Obj = P0 | P1 | P2 deriving (Eq,Ord,Show)
-- data Atom = At1 | At2 | At3 | At4 deriving (Eq,Ord,Show)
```

together with the spelled-out definitions of the translated atom category definition.

Now the original atom category definition is given by the following expression:

```
-- ra_acat (raMat raB [[],[()],[(),()]]) :: ACat (Vec ()) (MatMor () Bool)
```

Its object and morphism types are already instances of the class `Show`, so we cannot rely on the functions provided by this class, but have to explicitly provide the corresponding output functions.

For efficiency, we always use functions of the prelude type `ShowS`.

For enabling to generate such definitions also by other means, we generally also provide intermediate functions that do not expect a full atom category definition.

The first component we need is the mapping from pairs of objects to atomsets. We introduce a default definition with the empty atomset as result, so we need not explicitly output those mappings that do have the empty atomset as result:

```
showsAtomset0 :: ShowS -> (obj -> ShowS) -> (atom -> ShowS) ->
  [obj] -> (obj -> obj -> [atom]) -> ShowS
showsAtomset0 indent so sa objects atomset = ffold (do
   x <- objects
   y <- objects
   case atomset x y of
    [] -> []
    atoms -> [indent . ("atomset " ++) . so x . (' ' :) . so y . (" = " ++) .
                                  listShows sa atoms . ('\n' :)]
  ) . indent . ("atomset _ _ = []\n" ++)

showsAtomset' :: ShowS -> (obj -> ShowS) -> (atom -> ShowS)
                       -> ACat obj atom -> ShowS
showsAtomset' indent so sa ac =
  showsAtomset0 indent so sa (acat_objects ac) (acat_atomset ac)

showsAtomset :: (Show obj, Show atom) => ShowS -> ACat obj atom -> ShowS
showsAtomset indent ac = showsAtomset' indent shows shows ac
```

Since we may generate atom composition table output not only directly from a given atom category definition, but also from cycle representations (see below), we provide separate access to the basic output functions:

```
showsAtCompEntry0 :: (at -> ShowS) -> ShowS -> at -> at -> [at] -> ShowS
showsAtCompEntry0 sa prefix a b c =
  prefix . sa a . (' ' :) . sa b . (" = " ++) . listShows sa c . ('\n' :)

showsAtCompDefault :: ShowS
showsAtCompDefault = ("atComp _ _ _  _ _ = []\n" ++)


showsAtCompEntry1 :: (obj -> ShowS) -> (atom -> ShowS) ->
                     obj -> obj -> obj -> atom -> atom -> [atom] -> ShowS
showsAtCompEntry1 so sa x y z a b c =
   showsAtCompEntry0 sa (atCompPrefix so x y z) a b c

atCompPrefix so x y z =
  ("atComp " ++) . listShowsSep so ' ' [x,y,z] . (' ' :)
```

Normally, we assume the equivalents of `acat_objects`, `acat_atomset` and `acat_comp` to be available:

```
showsAtComp0 :: ShowS -> (obj -> ShowS) -> (atom -> ShowS) ->
  [obj] -> (obj -> obj -> [atom]) ->
  (obj -> obj -> obj -> atom -> atom -> [atom]) -> ShowS
showsAtComp0 indent so sa objects atomset comp = ffold (do
   x <- objects
   y <- objects
   z <- objects
   let prefix = indent . atCompPrefix so x y z
   let cmp = comp x y z
   a <- atomset x y
   b <- atomset y z
   case cmp a b of [] -> []
                   c  -> [showsAtCompEntry0 sa prefix a b c]
  ) . indent . showsAtCompDefault
```

Usually these are indeed taken from an atom category definition, and we also provide a variant that uses existing `Show` instances:

```
showsAtComp' :: ShowS -> (obj -> ShowS) -> (atom -> ShowS)
                     -> ACat obj atom -> ShowS
showsAtComp' indent so sa ac =
  showsAtComp0 indent so sa (acat_objects ac) (acat_atomset ac) (acat_comp ac)

showsAtComp :: (Show obj, Show atom) => ShowS -> ACat obj atom -> ShowS
showsAtComp indent ac = showsAtComp' indent shows shows ac
```

It is essentially the same story for the identity morphism:

```
showsIdmor0 :: ShowS -> (obj -> ShowS) -> (atom -> ShowS) ->
  [obj] -> (obj -> [atom]) -> ShowS
showsIdmor0 indent so sa objects idmor = ffold (do
   x <- objects
   case idmor x of
    [] -> []
    atoms -> [indent . ("idmor " ++) . so x . (" = " ++) .
                                  listShows sa atoms . ('\n' :)]
  ) . indent . ("idmor _ = []\n" ++)

showsIdmor' :: ShowS -> (obj -> ShowS) -> (atom -> ShowS)
                                    -> ACat obj atom -> ShowS
showsIdmor' indent so sa ac =
  showsIdmor0 indent so sa (acat_objects ac) (acat_idmor ac)

showsIdmor :: (Show obj, Show atom) => ShowS -> ACat obj atom -> ShowS
showsIdmor indent ac = showsIdmor' indent shows shows ac
```

All these together are now used to output a complete atom category definition with the components defined locally in a `where` clause:

```
showsACat0 :: String -> ShowS -> (obj -> ShowS) -> (atom -> ShowS) ->
  [obj] -> (obj -> obj -> [atom]) ->
  (obj -> obj -> obj -> atom -> atom -> [atom]) -> (obj -> [atom]) -> ShowS
showsACat0 name indent so sa objects atomset comp idmor =
 let indent' = indent . ("  " ++) in
  indent . ("aCat_" ++) . (name ++) . (" = ACat\n" ++) .
  indent . ("  {acat_isObj   = (`elem` objects)\n" ++) .
  indent . ("  ,acat_isAtom  = (\\ s t a -> a `elem` atomset s t)\n" ++) .
  indent . ("  ,acat_objects = objects\n" ++) .
  indent . ("  ,acat_atomset = atomset\n" ++) .
  indent . ("  ,acat_idmor   = idmor\n" ++) .
  indent . ("  ,acat_comp    = atComp\n" ++) .
  indent . ("  }\n\n" ++) .
  indent . (" where\n" ++) .
  indent' . ("objects = " ++) . listShows so objects . ("\n\n" ++) .
  showsAtomset0 indent' so sa objects atomset . ('\n' :) .
  showsAtComp0 indent' so sa objects atomset comp .('\n' :) .
  showsIdmor0 indent' so sa objects idmor


showsACat' :: String -> ShowS -> (obj -> ShowS) -> (atom -> ShowS)
                    -> ACat obj atom -> ShowS
showsACat' name indent so sa ac =
  showsACat0 name indent so sa (acat_objects ac) (acat_atomset ac)
                               (acat_comp ac)    (acat_idmor ac)

showsACat :: (Show obj, Show atom) => String -> ShowS -> ACat obj atom -> ShowS
showsACat name indent ac = showsACat' name indent shows shows ac
```

For the converse table, we collect identical mappings into the default case. For this purpose we should not compare original atomic morphisms, but their string representation, since usually the same atom output name may occur in different atom sets, but atomic morphisms from different homsets are always different.

```
showsConv0 :: ShowS -> (obj -> ShowS) -> (atom -> ShowS) ->
  [obj] -> (obj -> obj -> [atom]) ->
  (obj -> obj -> atom -> atom) -> ShowS
showsConv0 indent so sa objects atomset conv = ffold (do
   x <- objects
   y <- objects
   let cnv = conv x y
   a <- atomset x y
   let c = cnv a
   if sa c "" == sa a ""
    then []
    else [indent . ("conv " ++) . so x . (' ' :) . so y . (' ' :) .
                           sa a . (" = " ++) . sa c . ('\n' :)]
  ) . indent . ("conv _ _  x = x\n" ++)
```

So far, the code might just as well be used for allegories, since the type of converse there is the same as in atom allegory descriptions.

But now we provide direct output only for the converse functions of atom allegory descriptions:

```
showsAtConv' :: ShowS -> (obj -> ShowS) -> (atom -> ShowS) ->
                AAll obj atom -> ShowS
showsAtConv' indent so sa aa =
  showsConv0 indent so sa (aall_objects aa) (aall_atomset aa) (aall_converse aa)

showsAtConv :: (Show obj, Show atom) => ShowS -> AAll obj atom -> ShowS
showsAtConv indent aa = showsAtConv' indent shows shows aa
```

When writing an atom allegory definition, we first output the atom category definition contained within it, and then put the `AAll` definition on the same level, again with the converse table as a local definition:

```
showsAAll0 :: String -> ShowS -> (obj -> ShowS) -> (atom -> ShowS) ->
  [obj] -> (obj -> obj -> [atom]) ->
  (obj -> obj -> obj -> atom -> atom -> [atom]) -> (obj -> [atom]) ->
  (obj -> obj -> atom -> atom) -> ShowS
showsAAll0 name indent so sa objects atomset comp idmor conv =
 let indent' = indent . ("  " ++) in
  showsACat0 name indent so sa objects atomset comp idmor . ('\n' :) .
  indent . ("aAll_" ++) . (name ++) . (" = AAll\n" ++) .
  indent . ("  {aall_acat    = aCat_" ++) . (name ++) . ('\n' :) .
  indent . ("  ,aall_converse = conv\n" ++) .
  indent . ("  }\n\n" ++) .
  indent . (" where\n" ++) .
  showsConv0 indent' so sa objects atomset conv

showsAAll' :: String -> ShowS -> (obj -> ShowS) -> (atom -> ShowS) ->
              AAll obj atom -> ShowS
showsAAll' name indent so sa ac =
  showsAAll0 name indent so sa (aall_objects ac) (aall_atomset ac)
             (aall_comp ac) (aall_idmor ac) (aall_converse ac)

showsAAll :: (Show obj, Show atom) => String -> ShowS -> AAll obj atom -> ShowS
showsAAll name indent ac = showsAAll' name indent shows shows ac
```

We round this off with functions that in addition output the definition of the atom set relation algebra on the same level as the other two definitions:

```
showsARA0 :: String -> ShowS -> (obj -> ShowS) -> (atom -> ShowS) ->
  [obj] -> (obj -> obj -> [atom]) ->
  (obj -> obj -> obj -> atom -> atom -> [atom]) -> (obj -> [atom]) ->
  (obj -> obj -> atom -> atom) -> ShowS
showsARA0 name indent so sa objects atomset comp idmor conv =
 let indent' = indent . ("  " ++) in
  showsAAll0 name indent so sa objects atomset comp idmor conv . ('\n' :) .
  indent . ("ra_" ++) . (name ++) .
          (" = atomsetRA aAll_" ++) . (name ++) . ('\n' :)
```

```
showsARA' :: String -> ShowS -> (obj -> ShowS) -> (atom -> ShowS) ->
             AAll obj atom -> ShowS
showsARA' name indent so sa ac =
  showsARA0 name indent so sa (aall_objects ac) (aall_atomset ac)
            (aall_comp ac) (aall_idmor ac) (aall_converse ac)

showsARA :: (Show obj, Show atom) => String -> ShowS -> AAll obj atom -> ShowS
showsARA name indent ac = showsARA' name indent shows shows ac
```

2.4.8 Generating Atom Set Definitions for Boolean Matrix Algebras

For experiments, we want to output atom descriptions for algebras of Boolean matrices — if a Boolean matrix is an atom, we can compute its ordinal number in a natural ordering of atomic matrices of this shape with the following function:

```
boolMatAtomPos :: [[Bool]] -> Int
boolMatAtomPos = fst . head . filter snd . zip [1..] . concat
```

This allows us to define `shows` functions for objects and atoms of algebras in the range of `distrAllMat distrAllB`:

```
boolMatAtomName i = "At" ++ show i
boolMatAtomShows mm = let (m,_,_) = unMatMor mm
                      in ((boolMatAtomName $ boolMatAtomPos m) ++)

boolMatObjShows obj = ('P' :) . shows (length $ unVec obj)
```

The following function displays the composition tables of atom category definitions for algebras of Boolean matrices:

```
boolMatAtCompSchows indent objs =
   showsAtComp' indent boolMatObjShows boolMatAtomShows $
   distrAll_acat $ distrAllMat distrAllB objs
```

For example, the following invocation prints the composition table of Boolean 1×1, 1×2, 2×1, and 2×2 matrices:

```
putStr $ boolMatAtCompSchows id [[()],[(),()]] ""
```

Whole algebras can be printed with the following:

```
boolMatACatSchows :: String -> ShowS -> [[()]] -> ShowS
boolMatACatSchows name indent objs =
   showsACat' name indent boolMatObjShows boolMatAtomShows $
   distrAll_acat $ distrAllMat distrAllB objs

boolMatAAllSchows :: String -> ShowS -> [[()]] -> ShowS
boolMatAAllSchows name indent objs =
   showsAAll' name indent boolMatObjShows boolMatAtomShows $
   distrAll_aall $ distrAllMat distrAllB objs
```

When writing whole relation algebras of this form, we also include the allegory representations between the newly generated atom set relation algebra and the corresponding Boolean matrix relation algebra (see 2.4.11), including also an equivalence test. For this purpose we also generate definitions of a few local auxiliary functions, which are, however, not exported from the resulting module.

```
boolMatARASchows :: String -> ShowS -> [[()]] -> ShowS
boolMatARASchows name indent objlist =
   indent . ("module " ++) . (name ++) .
   ('(' :) . listShowsSep (++) ',' (map (++ name)
                ["aCat_","aAll_","ra_","matBtoAtCat_","atCatToMatB_", "atMat_",
                 "raB_","allB_","test_for_equivalence_"]) .
   (") where\n\n" ++) .
   indent . ("import RelAlg \n\n" ++) .
   indent . ("import Matrix \n\n" ++) .
   indent . ("import Atomset \n\n" ++) .
   indent . mkdata ("Obj" ++ name) (map (flip boolMatObjShows "") objects) .
   indent . mkdata ("Atom" ++ name)
     (map (flip boolMatAtomShows "") $
          (do x <- objects; y <- objects; aall_atomset aall x y)) .
   showsARA' name indent boolMatObjShows boolMatAtomShows aall .
   ('\n' :) .
   ((do x <- objs
        indent "vecToObj " ++ show x ++ " = "
                           ++ boolMatObjShows (vec x) "\n") ++) .
   ('\n' :) .
   ((do x <- objs
        indent "objToVec " ++ boolMatObjShows (vec x) (" = "
                           ++ show x ++ "\n")) ++)
   . ('\n' :) .
   ffold (do x <- objs
             let lx = length x
             y <- objs
             let ly = length y
             [indent . ("atMat_" ++) . (name ++) . (' ' :) .
                       boolMatObjShows (vec x) . (' ' :) .
                       boolMatObjShows (vec y) . (" = " ++) .
              listShows (listShows ((++) . boolMatAtomName))
                        (take lx $ unfold (splitAt ly) [1..]) . ('\n' :)]
         ) . ('\n' :) .
   indent . ("matBtoAtCat_" ++) . (name ++) .
```

```
            (" = matBtoAtCat vecToObj atMat_" ++) . (name ++) . ("\n\n" ++) .
   indent . ("atCatToMatB_" ++) . (name ++) .
            (" = atCatToMatB objToVec atMat_" ++) . (name ++) . ("\n\n" ++) .
   indent . (("raB_" ++ name ++ " = raMat raB " ++ show objs) ++) .
   ('\n' :) .
   indent . (("allB_" ++ name ++ " = ra_all raB_" ++ name) ++) .
   ("\n\n" ++) .
   indent . (("test_for_equivalence_" ++ name ++ " =\n" ++
              indent " all_equiv_perform allB_" ++ name ++
              "(ra_all ra_" ++ name ++ ") matBtoAtCat_" ++ name ++
              " atCatToMatB_" ++ name) ++) . ('\n' :)
 where mkdata name cs = ("data " ++) . (name ++) . (" = " ++) .
                        ((foldr1 (\l r -> l ++ " | " ++ r)
                                 (foldr insertSet [] cs)) ++) .
                        (" deriving (Eq, Ord, Show)\n\n" ++)
       aall = distrAll_aall $ distrAllMat distrAllB objs
       objects = aall_objects aall
       objs = foldr insertSet [] objlist
```

Finally, we wrap this into a function that generates a file containing a literate Haskell module, adhering to the usual naming convention:

```
writeBoolMatARA' :: String -> [[()]] -> IO ()
writeBoolMatARA' name objs =
  writeFile (name ++ ".lhs")
  (("This file has been automatically generated.\n\n" ++) $
   ("It contains a description of a Boolean matrix relation algebra\n" ++) $
   ("expressed in terms of its atoms.\n\n\n" ++) $
   boolMatARASchows name ("> " ++) objs ""
  )

writeBoolMatARA :: String -> [Int] -> IO ()
writeBoolMatARA name objs =
   writeBoolMatARA' name $ map (flip replicate ()) $ filter (0 <=) objs
```

Generating an atom definition of a matrix algebra now boils down to typing the following command in Hugs:

```
-- writeBoolMatARA "Q012" [0,1,2]
```

The resulting Haskell source file "`Q012.lhs`" can then immediately be used.

2.4.9 Cycles

A *cycle* is a triple of atoms `at1`, `at2`, `at3` such that `at3` occurs in `at1 ‘comp‘ at2`. This will then mean that also `at2` occurs in `(conv at1) ‘comp‘ at3`

`(conv at1)` occurs in `at2 ‘comp‘ (conv at3)`

`(conv at3)` occurs in `(conv at2) ‘comp‘ (conv at1)`

`(conv at2)` occurs in `(conv at3) ‘comp‘ at1`

`at1` occurs in `at3 ‘comp‘ (conv at2)`

This is easily proved starting from the assumption that this might not be true and using the properties of atoms.

As this reduces the number of composition table entries, it may sometimes be used to shorten atom composition definitions.

For our `Cycle` data-type, we include all three involved objects along with the three atoms:

```
type Cycle obj atom = ((obj,obj,obj),(atom,atom,atom))
```

In order to determine the list of cycles, we first generate all triples of atoms and transposed atoms with the respective composition property in a list and then cancel them in groups of (at most) 6.

```
compTriples aa = let
    objects = aall_objects aa
    atoms = aall_atomset aa
    atComp = aall_comp aa
 in [ ((x, y, z), (a, b, c))  |
           x <- objects,    y <- objects,    z <- objects,
           a <- atoms x y,  b <- atoms y z,  c <- atComp x y z a b ]


cycles aa = let conv = aall_converse aa
      in nub [ ((x, y, z), sort [(a, b, c),
                                 (conv x y a, c, b),
                                 (b, conv x z c, conv x y a),
                                 (conv y z b, conv x y a, conv x z c),
                                 (conv x z c, a, conv y z b),
                                 (c, conv y z b, a)])
             |  ((x, y, z), (a, b, c)) <- compTriples aa
             ]

cycleRepresentatives :: (Ord atom, Eq obj) =>
                        AAll obj atom -> [Cycle obj atom]
cycleRepresentatives aa = map (\ (x, y) -> (x, head y)) $ cycles aa
```

We now recompute the atom composition tables from the cycle representatives. The first step is to expand a cycle into the six atom triples it represents:

```
oneCycle conv ((x, y, z), (a, b, c)) =
   [((x, y, z), (a,         b,         c        )),
    ((y, x, z), (conv x y a, c,         b        )),
    ((y, z, x), (b,         conv x z c, conv x y a)),
    ((z, y, x), (conv y z b, conv x y a, conv x z c)),
    ((z, x, y), (conv x z c, a,         conv y z b)),
    ((x, z, y), (c,         conv y z b, a        ))
   ]
```

We use a nested finite map for storing the composition information:

```
type AtomCompTable obj atom =
     FiniteMap (obj,obj,obj) (FiniteMap (atom,atom) (Set atom))



allCycles :: (Ord obj, Ord atom) => (obj -> obj -> atom -> atom) ->
                                    [Cycle obj atom] -> AtomCompTable obj atom
allCycles conv cycs = foldr addCycle zeroFM $ concatMap (oneCycle conv) cycs

addCycle (objs,ats) fm = let atfm = lookupDftFM fm zeroFM objs
                         in addToFM objs (addCyc ats atfm) fm

addCyc (a,b,c) fm = let
  p = (a,b)
  s = lookupDftFM fm zeroSet p
 in addToFM p (addToSet c s) fm
```

Using the following functions, such a table can be used to directly define an atom composition function, or to define one as the table's complement wrt. an atom supply that has to be passed as another argument:

```
tableAtComp :: (Ord obj, Ord atom) =>
               AtomCompTable obj atom ->
               obj -> obj -> obj -> atom -> atom -> [atom]
tableAtComp atct x y z a b = let
   atfm = lookupDftFM atct zeroFM  (x,y,z)
   cs   = lookupDftFM atfm zeroSet (a,b)
  in toListSet cs

negTableAtComp :: (Ord obj, Ord atom) =>
               (obj -> obj -> [atom]) ->
               AtomCompTable obj atom ->
               obj -> obj -> obj -> atom -> atom -> [atom]
negTableAtComp atomset atct = let
   neg = negAtCompTable atomset atct
 in \ x y z -> let
   atoms = atomset x z
 in \ a b -> case lookupFM neg (x,y,z) of
     Nothing -> atoms
     Just atfm -> case lookupFM atfm (a,b) of
                    Nothing -> atoms
                    Just cs -> toListSet cs

-- not exported, since not independently useable!
negAtCompTable :: (Ord obj, Ord atom) =>
               (obj -> obj -> [atom]) ->
               AtomCompTable obj atom -> AtomCompTable obj atom
negAtCompTable atomset = mapFM (\ (x,y,z) ->
    let atoms = listToSet $ atomset x z
    in mapFM (\ (a,b) cs -> atoms `diffSet` cs))
```

Finally we present a function that allows to print an explicit variant of the atom composition table that results from a cycle list:

```
showsCycAtComp :: (Ord obj, Ord atom) => (obj -> ShowS) -> (atom -> ShowS) ->
  (obj -> obj -> atom -> atom) -> [Cycle obj atom] -> ShowS
showsCycAtComp so sa conv cycs s =
  foldFM (\ (x,y,z) atfm s0 ->
   foldFM (\ (a,b) cs -> showsAtCompEntry1 so sa x y z a b (toListSet cs))
          s0 atfm)
--  foldr (\ (x, y, z, a, b, c) -> showsAtCompEntry1 so sa x y z a b c)
        (showsAtCompDefault s) (allCycles conv cycs)
-- (cycleAtCompTable conv cycs)
```

2.4.10 Building Atom Category Definitions from Distributive Allegories

We can extract atom category and atom allegory definitions from a distributive allegory. Note that this decribes a relation algebra, but in general only one that is embedded (as a distributive allegory) in the original distributive allegory.

```
distrAll_acat :: (Ord obj,Eq mor) => DistrAll obj mor -> ACat obj mor
distrAll_acat da =
 let objects = distrAll_objects da
     objDom = listToFM (zip objects (repeat ()))
     memoObj = memoFMfm' objDom
     objPairs = do a <- objects; b <- objects; [(a,b)]
     objPairDom = listToFM (zip objPairs (repeat ()))
     memoObjPair = curry . memoFMfm' objPairDom . uncurry
     atoms = distrAll_atoms da
 in ACat
  {acat_isObj    = distrAll_isObj da
  ,acat_isAtom   = distrAll_isAtom da
  ,acat_objects = distrAll_objects da
  ,acat_atomset = memoObjPair (distrAll_atomset da)
  ,acat_idmor   = memoObj (atoms . distrAll_idmor da)
  ,acat_comp    = (\ _ _ _  f g -> atoms (distrAll_comp da f g))
  }

distrAll_aall :: (Ord obj,Eq mor) => DistrAll obj mor -> AAll obj mor
distrAll_aall da = AAll
  {aall_acat = distrAll_acat da
  ,aall_converse = (\ _ _ at -> distrAll_converse da at)
  }
```

For ease of access, we provide this interface also for the higher structures:

```
divAll_acat :: (Ord obj,Eq mor) => DivAll obj mor -> ACat obj mor
divAll_acat = distrAll_acat . divAll_distrAll

divAll_aall :: (Ord obj,Eq mor) => DivAll obj mor -> AAll obj mor
divAll_aall = distrAll_aall . divAll_distrAll

ded_acat :: (Ord obj,Eq mor) => Ded obj mor -> ACat obj mor
ded_acat = distrAll_acat . ded_distrAll

ded_aall :: (Ord obj,Eq mor) => Ded obj mor -> AAll obj mor
ded_aall = distrAll_aall . ded_distrAll

ra_acat :: (Ord obj,Eq mor) => RA obj mor -> ACat obj mor
ra_acat = distrAll_acat . ra_distrAll

ra_aall :: (Ord obj,Eq mor) => RA obj mor -> AAll obj mor
ra_aall = distrAll_aall . ra_distrAll
```

2.4.11 Equivalence for Matrix Atom Set Descriptions

For Boolean matrices, we provide a fast way to define the functors between the Boolean matrix algebra and an equivalent atom set algebra. This is used in the atom allegory definitions for Boolean matrix relation algebras that are generated in 2.4.7. The most important argument needed here is a function mapping pairs of objects to their *atom matrix*, which is a matrix of the shape of the corresponding Boolean matrix, but containing in every position that atom that stands for the atomic matrix with `True` at that position:

```
matBtoAtCat :: Ord atom => ([t] -> obj) -> (obj -> obj -> [[atom]]) ->
                           Fun obj (SetMor obj atom) (Vec t) (MatMor t Bool)
matBtoAtCat bToPobj atMat =
  let fmor m = let (bm,s,t) = unMatMor m
                   s' = bToPobj s
                   t' = bToPobj t
                   enter b a = if b then addToSet a else id
                   am = atMat s' t'
                   as = ffold (concat $ matZipWith enter bm am) zeroSet
               in SetMor (as,s',t')
  in Fun (bToPobj . unVec) fmor

atCatToMatB :: Ord atom => (obj -> [t]) -> (obj -> obj -> [[atom]]) ->
                           Fun (Vec t) (MatMor t Bool) obj (SetMor obj atom)
atCatToMatB pToBobj atMat =
  let fmor (SetMor (as,s,t)) =
         matMor (matMap (`elemSet` as) (atMat s t))
                (pToBobj s) (pToBobj t)
  in Fun (vec . pToBobj) fmor
```

2.4.12 Matrix Atom Category Definitions

With the tools available so far, we already can build matrix algebras over atom set algebras. But such a matrix algebra again has atoms — one might extract them using the tools from the previous section.

In this section we provide a direct definition of atom algebra descriptions for matrix algebras over atom set algebras.

An atomic matrix is a matrix where one coefficient is an atom, and all other coefficients are zero morphisms, i.e., empty sets. Therefore, an atomic matrix is described by the single atom it contains together with the position where it contains that atom:

```
type MatAt obj atom = (Int,Int,atom)
```

Working with matrix atoms turns out to be a lot easier than working with matrices:

```
acatMat :: Eq obj => ACat obj atom -> [[obj]] -> ACat [obj] (MatAt obj atom)
acatMat c oss = let objects = nub oss
 in if not (all (all (acat_isObj c)) oss) then error "acatMat: non-objects"
 else ACat
  {acat_isObj   = (`elem` objects)
  ,acat_isAtom  = (\ as bs (i,j,at) ->
      i >= 0 && i < length as &&
      j >= 0 && j < length bs &&
      let a = as !! i
          b = bs !! j
      in acat_isAtom c a b at)
  ,acat_objects = objects
  ,acat_atomset = (\ as bs ->
                     do (a,i) <- zip as [0..]
                        (b,j) <- zip bs [0..]
                        at <- acat_atomset c a b
                        return (i,j,at))
  ,acat_idmor   = (\ as ->
                     do (a,i) <- zip as [0..]
                        at <- acat_idmor c a
                        return (i,i,at))
  ,acat_comp    = (\ as bs cs (i1,j1,at1) (i2,j2,at2) ->
                     if j1 /= i2 then []
                     else let ats = acat_comp c (as !! i1) (bs !! i2) (cs !! j2)
                                                at1 at2
                          in map (\ at -> (i1,j2,at)) ats)
  }

aallMat :: Eq obj => AAll obj atom -> [[obj]] -> AAll [obj] (MatAt obj atom)
aallMat c oss = AAll
  {aall_acat = acatMat (aall_acat c) oss
  ,aall_converse = (\ as bs (i,j,at) ->
                      (j,i,aall_converse c (as !! i) (bs !! j) at))
  }
```

2.4.13 Example Atom Sets

For use in constructing relation algebras where the atoms do not carry clear-cut identities we offer a few finite sets, so that redefinitions and clashing exports are easier to avoid.

The following are therefore separate Haskell modules in separate source files:

```
module A2 where

data A2 = At1 | At2  deriving (Eq, Ord, Show, Read)
atomsA2 = [At1, At2]
```

```
module A4 where

data A4 = At1 | At2 | At3 | At4  deriving (Eq, Ord, Show, Read)
atomsA9 = [At1, At2, At3, At4]
```

```
module A9 where

data A9 = At1 | At2 | At3 | At4 | At5 | At6 | At7 | At8 | At9
   deriving (Eq, Ord, Show, Read)
atomsA9 = [At1, At2, At3, At4, At5, At6, At7, At8, At9]
```

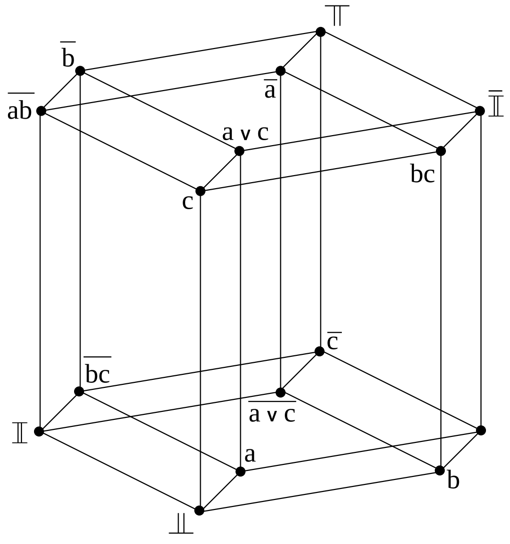
Chapter 3

Non-Standard Relation Algebras

Using the constructions of Chapters 1 and 2, we now present examples of relation algebras - some of them with quite unexpected properties. Firstly, there are relation algebras which, considered from the classical viewpoint, fail to correspond to our imagination, e.g., the McKenzie-, Maddux-Algebra, and also non-uniform relation algebras. Secondly, there are relation algebras to model quite simple everyday situations such as compass directions, interval interdependency, to model spatial information with "mereology", etc. Thirdly, we seem to be able to model strictly parallel net-like situations as with LRNnoc. This means explicitly excluding the attitude that "one might observe a net situation in full detail — at least in principle".

3.1 The McKenzie Relation Algebra

A nice non-representable relation algebra is the McKenzie algebra [McK70]. The following explanation is simply recalled from [SS89, SS93].



Boolean lattice structure of the McKenzie algebra

There are four atoms which might be called generators, namely $\mathbb{I}, a, b, c$; they are the upper neighbors of $\mathbb{L}$. Each of the 16 elements can be expressed as a subset of these four, so that a boolean lattice is established giving a four-dimensional cube. So $\mathbb{L}$ corresponds to the union of none and $\mathbb{T}$ to the union of all of them, a to the union of just one, and so forth. For instance, $\overline{a}$ means that all but a are united.

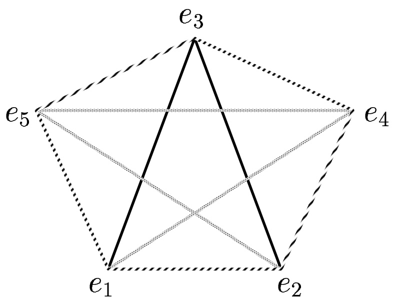
Transposition and composition is defined by giving the set

$$\mathbb{I}^\smile = \mathbb{I}, \quad a^\smile = c, \quad b^\smile = b, \quad c^\smile = a, \quad a^2 = a, \quad b^2 = \overline{b}, \quad c^2 = c,$$

$$a{;}c = c{;}a = \top, \quad a{;}b = b{;}a = a \sqcup b, \quad c{;}b = b{;}c = c \sqcup b$$

of basic compositions, from which the rest of compositions can be elaborated by distributivity.

Assume now that there exists a representation for the elements of this relation algebra in a set of 16 Boolean $n \times n$-matrices. We will prove that this assumption leads to a contradiction, so that such a representation cannot exist. As a satisfies $a^2 = a \sqsubseteq \overline{\mathbb{I}}$ and $a \sqcap a^{\smile} = a \sqcap c = \bot\!\!\!\bot$, it corresponds to a transitive and irreflexive matrix which is therefore a strict-ordering. By definition, c is the converse of a, so that b gives the relationship of incomparability with respect to this strict-ordering a since $\overline{b} = \mathbb{I} \sqcup a \sqcup c$. From $a{;}a^{\smile} = \top$ it follows that any pair of elements has a common upper bound which is different from both of them. On the other hand, $\overline{b} = b^2$ demands that any pair of comparable elements has some element to which they are both incomparable.



Case analysis

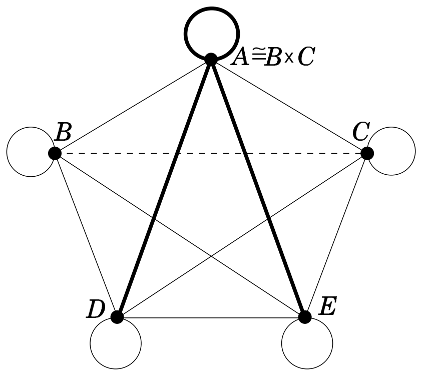
Now we switch to the usual notation $<$ for the strict-ordering a. This is the crucial step. We are no longer satisfied with just composition tables as before, but are looking for some set together with a strict-ordering satisfying the properties just mentioned. Since $b \neq \bot\!\!\!\bot$, there exist at least two incomparable elements e_1, e_2 in the set. They have some common upper bound e_3 with $e_1 < e_3, e_2 < e_3$, so that elements e_4, e_5 must exist with all the pairs $(e_1, e_5), (e_3, e_5), (e_2, e_4), (e_3, e_4)$ incomparable. However, from consecutive incomparability we may then conclude that $(e_2, e_5), (e_1, e_4), (e_4, e_5)$ are comparable pairs of elements. Now a case analysis leads to a contradiction: First we have that $e_1 < e_4, e_2 < e_5$, since e_4, e_5 must not be less than e_3. Then $e_4 \neq e_5$ since, by assumption, e_1, e_2 are incomparable which would contradict $(e_1, e_5), (e_2, e_4)$ (with $e_4 = e_5$) being incomparable. In this situation there is no possibility to decide whether $e_4 < e_5$ or $e_5 < e_4$, without introducing the forbidden relationships $e_1 < e_5$ or $e_2 < e_4$ by transitivity.

The translation of the algebraic description of the McKenzie algebra into Haskell is completely straightforward:

```
module McKenzie(aCat_McKenzie,aAll_McKenzie,ra_McKenzie) where

import RelAlg
import Atomset

aCat_McKenzie :: ACat () Atom
aCat_McKenzie = ACat
  {acat_isObj   = const True
  ,acat_isAtom  = (\ _ _  _ -> True)
  ,acat_objects = [()]
  ,acat_atomset = const $ const [I, A, B, C]
  ,acat_idmor   = const [I]
  ,acat_comp    = (\ _ _ _  -> comp)
  }

aAll_McKenzie :: AAll () Atom
aAll_McKenzie = AAll
  {aall_acat = aCat_McKenzie
  ,aall_converse = (\ _ _ -> conv)
  }

ra_McKenzie :: RA () (SetMor () Atom)
ra_McKenzie = atomsetRA aAll_McKenzie


data Atom = I | A | B | C   deriving (Eq, Show, Ord)


conv I = I
conv A = C
conv C = A
conv B = B


comp I x = [x]
comp x I = [x]
comp A A = [A]
comp A B = [A, B]
comp B A = [A, B]
comp A C = [I, A, B, C]
comp C A = [I, A, B, C]
comp B C = [B, C]
comp C B = [B, C]
comp B B = [I, A, C]
comp C C = [C]
```

3.2 Maddux

This algebra originates from a discussion of the second named author with Roger Maddux on the occasion of the 1991 Stefan Banach Semester on Algebraic Logic in Warsaw. They discussed the possibility of existence of a relation algebra for which the sharpness equation fails to hold. On a workshop in Rio de Janeiro in 1995, Roger Maddux explained the idea for the following heterogeneous relation algebra with 5 objects `A, B, C, D, E` [Mad95].



The underlying graph of the category

The morphism sets are defined to have 16 (for the bold lines), 2 (for the dotted line) or — in all other cases — 4 morphisms.

For this heterogeneous relation algebra, one will find out that the top object is — as indicated — the direct product of those of the second level. The morphisms between these, however, are very restricted in number. As one may see using the Haskell programs, the relation algebra given in detail below shows that only

$$(\pi;P;\pi^{\smile} \sqcap \rho;R;\rho^{\smile});(\pi;Q;\pi^{\smile} \sqcap \rho;S;\rho^{\smile}) \quad \underset{\neq}{\sqsubseteq} \quad \pi;P;Q;\pi^{\smile} \sqcap \rho;R;S\rho^{\smile}$$

A computer-aided proof that this is indeed such an algebra, and that it is unsharp was finally given by Michael Winter. He developed an explicit atom composition table that is included in the distribution as module `MadduxOrig`. The proof is also possible by the present program.

```
module Maddux(aCat_Maddux,aAll_Maddux,ra_Maddux) where

import RelAlg
import Atomset
import A4

import FiniteMaps
import Properties


data MadduxObj = A | B | C | D | E  deriving (Eq, Ord, Show, Read)
objseq = [A, B, C, D, E]


aCat_Maddux :: ACat MadduxObj A4
aCat_Maddux = ac where
 ac = ACat
  {acat_isObj    = const True
  ,acat_isAtom   = (\ s t m -> m ‘elem‘ atoms s t)
  ,acat_objects  = objseq
  ,acat_atomset  = atoms
  ,acat_idmor    = (\ s -> [At1]) -- acat_idmor_defaultM ac
  ,acat_comp    = atComp
  }

aAll_Maddux :: AAll MadduxObj A4
aAll_Maddux = AAll
  {aall_acat = aCat_Maddux
  ,aall_converse = conv
  }

conv _ _ a = a

ra_Maddux :: RA MadduxObj (SetMor MadduxObj A4)
ra_Maddux = atomsetRA aAll_Maddux

atoms :: MadduxObj -> MadduxObj -> [A4]
atoms A B = [At1, At2]
atoms A C = [At1, At2]
atoms A _ = allatoms
atoms B A = [At1, At2]
atoms C A = [At1, At2]
atoms _ A = allatoms
atoms B C = [At1]
atoms C B = [At1]
atoms _ _ = [At1, At2]

allatoms = [At1, At2, At3, At4]
```

In [Mad95], composition is defined via forbidden cycles, and we directly carry over that definition into our notation, using `negTableAtComp` from 2.4.9:

```
atComp = negTableAtComp atoms (allCycles conv forbidden)

forbidden =
     objs (A,A,A) [(At2,At2,At3), (At2,At2,At4), (At2,At3,At3), (At3,At3,At4)]
  ++ objs (A,A,B) [(At2,At1,At2), (At3,At1,At1), (At4,At1,At1)]
  ++ objs (A,B,B) [(At1,At2,At1)]
  ++ objs (A,A,C) [(At2,At1,At1), (At3,At1,At2), (At4,At1,At1)]
  ++ objs (A,C,C) [(At1,At2,At1)]
  ++ objs (A,A,D) atsAAD
  ++ objs (A,B,D) ats017
  ++ objs (A,C,D) ats027
  ++ objs (A,A,E) atsAAD
  ++ objs (A,B,E) ats017
  ++ objs (A,C,E) ats027
  ++ objs (A,D,E) [(At1,At1,At1)]
  ++ [((s,s,t),(At1,a,b)) | s <- objseq , t <- objseq,
                            a <- atoms s t, b <- atoms s t, a /= b]
 where
```

```
  objs os = map (\ ats -> (os,ats))
  comb at as1 as2 = [(at,a,b) | a <- as1, b <- as2]
  atsAAD = comb At2 [At1,At2] [At3,At4] ++
           comb At3 [At1,At3] [At2,At4]
  ats017 = [(At1,At1,At3), (At1,At1,At4), (At1,At2,At1), (At1,At2,At2)]
  ats027 = [(At1,At1,At2), (At1,At1,At4), (At1,At2,At1), (At1,At2,At3)]
```

The list of products has exactly one element:

```
Main> ded_NonemptyProducts (ra_ded ra_Maddux)
[(B,C,A,SetMor ({At1},A,B),SetMor ({At1},A,C))]
```

From the evaluation of the expression

```
printAllTestResults $ distrAll_funTest $ ra_distrAll ra_Maddux
```

we see that the projections are the only two non-trivial functions in this relation algebra. There are two constellations (related via conversion symmetry) that show that this product is unsharp:

```
Main> let d = ra_ded ra_Maddux in
        performAll (ded_unsharp (head (ded_NonemptyProducts d))) d
=== Test Start ===
unsharpness example
 Objects:
  D
  E
 Morphisms:
  SetMor ({At1},D,B)
  SetMor ({At1},D,C)
  SetMor ({At1},B,E)
  SetMor ({At1},C,E)
  SetMor ({At2},D,E)
  SetMor ({At1, At2},D,E)
unsharpness example
 Objects:
  E
  D
 Morphisms:
  SetMor ({At1},E,B)
  SetMor ({At1},E,C)
  SetMor ({At1},B,D)
  SetMor ({At1},C,D)
  SetMor ({At2},E,D)
  SetMor ({At1, At2},E,D)
=== Test End   ===
```

3.3 Mereology

When representing spatial information in a data base, one will in the first place store rather local information in tabular form such as “object a touches object b” or “object a is contained in object b”. Questions will, however, ask for more complex concepts such as “Is a group of objects connected”, e.g. So reasoning on the basic entries takes place and the question has arisen as to logical basis of this reasoning.

Formalisation of such concepts goes back to Leśniewski and his mereology [Leś29]. Today quite a lot of papers has appeared, such as [BG91, Coh96, DSW99]. During these studies, several small models came up that proved to be relational algebras. This facilitates reasoning, as the bulk of well-known relational formulae is then available.

Today, mereology may be considered as a branch of research bringing topology to work in artificial intelligence methods in spatial reasoning.

The relation algebras of this chapter have been presented in a talk given by Ivo Düntsch on the occasion of the Seminar on Relational Methods in Computer Science (RelMiCS 4), in Warsaw in September 1998, cf. [DWM98], where they were presented as so-called “minimal algebras” and recognised as relation algebras by the second author of the current report. From this, a fruitful discussion and cooperation emerged, resulting not least in the paper [DSW99].

3.3.1 N1

The basic mereological qualifications to start with are just four, namely identity, *is-part-of*, and its converse as well as *disconnected*. And there is just one object `()` in the base category of the relation algebra N1, resulting in a homogeneous relation algebra.

We denote

1’	by `I`
p	by `P`
p#	by `PH` and
dc	by `DC`

to obtain correspondence with [DWM98].

```
module MereoN1(aCat_MereoN1,aAll_MereoN1,ra_MereoN1) where

import RelAlg
import Atomset
import Matrix

import FiniteMaps
import Sets

data Atomset = I | P | PH | DC   deriving (Eq, Ord, Show)
atoms :: [Atomset]
atoms = [I, P, PH, DC]

aCat_MereoN1 :: ACat () Atomset
aCat_MereoN1 = ac where
 ac = ACat
  {acat_isObj   = const True
  ,acat_isAtom  = (\ _ _ _ -> True)
  ,acat_objects = [()]
  ,acat_atomset = (\ _ _ -> atoms)
  ,acat_idmor   = acat_idmor_defaultM ac
  ,acat_comp    = (\ _ _ _ -> atComp)
  }

aAll_MereoN1 :: AAll () Atomset
aAll_MereoN1 = AAll
  {aall_acat = aCat_MereoN1
  ,aall_converse = (\ _ _ -> conv)
  }

ra_MereoN1 :: RA () (SetMor () Atomset)
ra_MereoN1 = atomsetRA aAll_MereoN1


conv :: Atomset -> Atomset
conv P  = PH
conv PH = P
conv x  = x


atComp :: Atomset -> Atomset -> [Atomset]

atComp I  I  = [I]
atComp I  P  = [P]
atComp I  PH = [PH]
atComp I  DC = [DC]

atComp P  I  = [P]
atComp P  P  = [P]
atComp P  PH = atoms
atComp P  DC = [DC]

atComp PH I  = [PH]
atComp PH P  = [I, P, PH]
atComp PH PH = [PH]
atComp PH DC = [PH, DC]

atComp DC I  = [DC]
atComp DC P  = [P, DC]
atComp DC PH = [DC]
atComp DC DC = atoms
```

3.3.2 C1

The first approach to mereology will now be refined by taking into account also that there may be *external contact* (ec) between two items. So a fifth atom is added. Much of the composition tables stays the same. One may say that a row and a column have to be added.

```
module MereoC1(aCat_MereoC1,aAll_MereoC1,ra_MereoC1) where

import RelAlg
import Atomset
import Matrix

import FiniteMaps
import Sets
```

The are five atoms are represented as follows:
1' by `I`, p by `P`, p# by `PH`, ec by `EC`, and dc by `DC`.

```
data Atomset = I | P | PH | EC | DC  deriving (Eq, Ord, Show)
atoms :: [Atomset]
atoms = [I, P, PH, EC, DC]

aCat_MereoC1 :: ACat () Atomset
aCat_MereoC1 = ac where
 ac = ACat
  {acat_isObj    = const True
  ,acat_isAtom   = (\ _ _ _ -> True)
  ,acat_objects  = [()]
  ,acat_atomset  = (\ _ _ -> atoms)
  ,acat_idmor    = acat_idmor_defaultM ac
  ,acat_comp     = (\ _ _ _ -> atComp)
  }

aAll_MereoC1 :: AAll () Atomset
aAll_MereoC1 = AAll
  {aall_acat = aCat_MereoC1
  ,aall_converse = (\ _ _ -> conv)
  }

ra_MereoC1 :: RA () (SetMor () Atomset)
ra_MereoC1 = atomsetRA aAll_MereoC1


conv :: Atomset -> Atomset
conv P  = PH
conv PH = P
conv x  = x
```

```
atComp :: Atomset -> Atomset -> [Atomset]

atComp I  I  = [I]
atComp I  P  = [P]
atComp I  PH = [PH]
atComp I  EC = [EC]
atComp I  DC = [DC]

atComp P  I  = [P]
atComp P  P  = [P]
atComp P  PH = [I, P, PH]
atComp P  EC = [EC, DC]
atComp P  DC = [DC]

atComp PH I  = [PH]
atComp PH P  = [I, P, PH]
atComp PH PH = [PH]
atComp PH EC = [EC]
atComp PH DC = [EC, DC]

atComp EC I  = [EC]
atComp EC P  = [EC]
atComp EC PH = [EC, DC]
atComp EC EC = [I, P, PH]
atComp EC DC = [PH]

atComp DC I  = [DC]
atComp DC P  = [EC, DC]
atComp DC PH = [DC]
atComp DC EC = [P]
atComp DC DC = [I, P, PH]
```

3.3.3 G

This is a third example of mereology. It is claimed by Düntsch (cf. [DWM98]) that, whenever a relational model for spatial reasoning is considered, then — under some other conditions — this algebra will be present. The interpretation of the algebra $\mathcal{G}$ is that we start with the concept

$$P \qquad \equiv \text{“is proper part of”}$$

and its converse $P^{\smile}$ and then consider

$$\begin{array}{lcll} O & \equiv & P^{\smile};P \sqcap \overline{1'} & \equiv \text{“being non-identical and properly overlapping”} \\ \# & \equiv & \overline{P \sqcup P^{\smile} \sqcup 1'} & \equiv \text{“being incomparable wrt. the (reflexive) is-part-of ordering”} \\ T & \equiv & P;P^{\smile} \sqcap \overline{1'} & \equiv \text{“being non-identical and commonly being properly topped”} \end{array}$$

The concept of O is then subdivided into $P, P^{\smile}, ON, OD$ and amended by DN, DD:

$$\begin{array}{lcl} ON & \equiv & O \sqcap \# \sqcap T \\ OD & \equiv & O \sqcap \# \sqcap \overline{T} \end{array}$$

$$DN \equiv \overline{\overline{O} \sqcap T}$$
$$DD \equiv \overline{O \sqcup T \sqcup 1'}$$

The interrelationship is then investigated taking the above as seven atoms, representing them in the following way:

1' by `I`	P by `P`	ON by `ON`	DN by `DN`
	$P^{\smile}$ by `PH`	OD by `OD`	DD by `DD`

```
module MereoG(aCat_MereoG,aAll_MereoG,ra_MereoG,assert_G_TEST) where

import RelAlg
import Atomset
import Matrix

import FiniteMaps
import Sets


data Atomset = I | P | PH | ON | OD | DN | DD  deriving (Eq, Ord, Show)
atoms = [I, P, PH, ON, OD, DN, DD]

aCat_MereoG :: ACat () Atomset
aCat_MereoG = ac where
 ac = ACat
  {acat_isObj   = const True
  ,acat_isAtom  = (\ _ _ _ -> True)
  ,acat_objects = [()]
  ,acat_atomset = (\ _ _ -> atoms)
  ,acat_idmor   = acat_idmor_defaultM ac
  ,acat_comp    = (\ _ _ _ -> atComp)
  }

aAll_MereoG :: AAll () Atomset
aAll_MereoG = AAll
  {aall_acat = aCat_MereoG
  ,aall_converse = (\ _ _ -> conv)
  }

ra_MereoG :: RA () (SetMor () Atomset)
ra_MereoG = atomsetRA aAll_MereoG

conv :: Atomset -> Atomset
conv P  = PH
conv PH = P
conv x  = x

atComp :: Atomset -> Atomset -> [Atomset]

atComp I  x  = [x]
atComp x  I  = [x]

atComp P  P  = [P]
atComp P  PH = [I, P, PH, ON, DN]
atComp P  ON = [P, ON, DN]
atComp P  OD = [P, ON, OD, DN, DD]
atComp P  DN = [DN]
atComp P  DD = [DN]

atComp PH P  = [I, P, PH, ON, OD]
atComp PH PH = [PH]
atComp PH ON = [PH, ON, OD]
atComp PH OD = [OD]
atComp PH DN = [PH, ON, OD, DN, DD]
atComp PH DD = [OD]

atComp ON P  = [P, ON, OD]
atComp ON PH = [PH, ON, DN]
atComp ON ON = [I, P, PH, ON, OD, DN, DD]
atComp ON OD = [P, ON, OD]
atComp ON DN = [PH, ON, DN]
atComp ON DD = [ON]

atComp OD P  = [OD]
atComp OD PH = [PH, ON, OD, DN, DD]
atComp OD ON = [PH, ON, OD]
atComp OD OD = [I, P, PH, ON, OD]
atComp OD DN = [PH]
atComp OD DD = [PH]

atComp DN P  = [P, ON, OD, DN, DD]
atComp DN PH = [DN]
atComp DN ON = [P, ON, DN]
atComp DN OD = [P]
atComp DN DN = [I, P, PH, ON, DN]
atComp DN DD = [P]

atComp DD P  = [OD]
atComp DD PH = [DN]
atComp DD ON = [ON]
atComp DD OD = [P]
atComp DD DN = [PH]
atComp DD DD = [I]
```

We now write some code to test the formulae of the introduction to this section:

```
gMor atoms = mkSetMor () () atoms
```

```
rI  = gMor [I ]
rP  = gMor [P ]
rPH = gMor [PH]
rON = gMor [ON]
rOD = gMor [OD]
rDN = gMor [DN]
rDD = gMor [DD]

(&&&) = ra_meet ra_MereoG
(^^^) = ra_comp ra_MereoG
compl = ra_compl ra_MereoG
(|||) = ra_join ra_MereoG

rO = (rPH ^^^ rP) &&& compl rI
rH = compl (rP ||| rPH ||| rI)
rT = (rP ^^^ rPH) &&& compl rI

assert_G_TEST :: [([Char],[()],[SetMor () Atomset])]
                     -> [([Char],[()],[SetMor () Atomset])]
assert_G_TEST =
  let rOmH = rO &&& rH
  in test (rON == (rOmH &&& rT)) [] [rON,rOmH &&& rT] "ON inconsistent" .
     let r = rOmH &&& compl rT in
     test (rOD == r) [] [rOD,r]  "OD inconsistent" .
     let r = compl rO &&& rT in
     test (rDN == r) [] [rDN,r] "DN inconsistent" .
     let r = compl (rO ||| (rT ||| rI)) in
     test (rDD == r) [] [rDD,r] "DD inconsistent" .
     test False [] [rO] "this is ``O'':" .
     test False [] [rH] "this is ``#'':" .
     test False [] [rT] "this is ``T'':"
```

The result is as expected:

```
Main> printAllTestResults assert_G_TEST
=== Test Start ===
this is ``O'':
Morphism: SetMor ({P, PH, ON, OD},(),())
this is ``#'':
Morphism: SetMor ({ON, OD, DN, DD},(),())
this is ``T'':
Morphism: SetMor ({P, PH, ON, DN},(),())
=== Test End   ===
```

3.4 An Interval Algebra

The following algebra [All81, AK83, vB83, MB83, All83] is well-known to specialists. A description may be found in the AMAST 1993 invited talk of Roger Maddux [Mad94].

```
module Interval(aCat_Interval,aAll_Interval,ra_Interval) where

import RelAlg
import Atomset
import FiniteMaps
```

For the interpretation of the algebra $\mathcal{IA}$ consider nonempty intervals on the real axis. They may be described by a pair of two different real numbers which we assume to be in ascending order. The first gives the starting time of the interval and the second the ending time. For simplicity, we assume only intervals $(x, y]$, i.e., left-open and right-closed.

On any given set of such intervals, we now consider the following relations together with their converses, if this is a different relation. Let the first interval be given by the pair $(x, x']$ and the second by $(y, y']$.

$1'$	$\equiv$	$x = y$ and $x' = y'$	$\equiv$ "identity of intervals"
p	$\equiv$	$x' < y$	$\equiv$ "first interval stricly precedes the second"
d	$\equiv$	$y < x < x' < y'$	$\equiv$ "first interval is bi-strictly contained in the second"
o	$\equiv$	$x < y < x' < y'$	$\equiv$ "first interval is partly overlapped by the second"
m	$\equiv$	$x < x' = y < y'$	$\equiv$ "first interval touches the second from the left"
s	$\equiv$	$x = y < x' < y'$	$\equiv$ "first interval is strict initial part of the second"
f	$\equiv$	$y < x < x' = y'$	$\equiv$ "first interval is strict terminal part of the second"

The interrelationship is then investigated by constructing a homogeneous relation algebra from a set of thirteen atoms induced by the above relations, using the following names:

`I`	for	$1'$	`P`	for	p	`Pc`	for	$p^{\smile}$	`M`	for	m	`Mc`	for	$m^{\smile}$
			`D`	for	d	`Dc`	for	$d^{\smile}$	`S`	for	s	`Sc`	for	$s^{\smile}$
			`O`	for	o	`Oc`	for	$o^{\smile}$	`F`	for	f	`Fc`	for	$f^{\smile}$

```
data Atomset = I | P | Pc | D | Dc | O | Oc | M | Mc | S | Sc | F | Fc
                     deriving (Eq, Ord, Show)
atoms = [I, P, Pc, D, Dc, O, Oc, M, Mc, S, Sc, F, Fc]

aCat_Interval :: ACat () Atomset
aCat_Interval = ac where
 ac = ACat
  {acat_isObj    = const True
  ,acat_isAtom   = (\ _ _ _ -> True)
  ,acat_objects  = [()]
  ,acat_atomset  = (\ _ _ -> atoms)
  ,acat_idmor    = acat_idmor_defaultM ac
  ,acat_comp     = (\ _ _ _ -> atComp)
  }

aAll_Interval :: AAll () Atomset
aAll_Interval = AAll
  {aall_acat = aCat_Interval
  ,aall_converse = (\ _ _ -> conv)
  }
```

```
ra_Interval :: RA () (SetMor () Atomset)
ra_Interval = atomsetRA aAll_Interval

conv :: Atomset -> Atomset
conv I  = I
conv P  = Pc
conv Pc = P
conv D  = Dc
conv Dc = D
conv O  = Oc
conv Oc = O
conv M  = Mc
conv Mc = M
conv S  = Sc
conv Sc = S
conv F  = Fc
conv Fc = F

atComp :: Atomset -> Atomset -> [Atomset]

atComp I x  = [x]
atComp x I  = [x]

atComp P P  = [P]
atComp P Pc = [I, P, Pc, D, Dc, O, Oc, M, Mc, S, Sc, F, Fc]
atComp P D  = [P, D, O, M, S]
atComp P Dc = [P]
atComp P O  = [P]
atComp P Oc = [P, D, O, M, S]
atComp P M  = [P]
atComp P Mc = [P, D, O, M, S]
atComp P S  = [P]
atComp P Sc = [P]
atComp P F  = [P, D, O, M, S]
atComp P Fc = [P]

atComp Pc P = [I, P, Pc, D, Dc, O, Oc, M, Mc, S, Sc, F, Fc]
atComp Pc Pc = [Pc]
atComp Pc D  = [Pc, D, Oc, Mc, F]
atComp Pc Dc = [Pc]
atComp Pc O  = [Pc, D, Oc, Mc, F]
atComp Pc Oc = [Pc]
atComp Pc M  = [Pc, D, Oc, Mc, F]
atComp Pc Mc = [Pc]
atComp Pc S  = [Pc, D, Oc, Mc, F]
atComp Pc Sc = [Pc]
atComp Pc F  = [Pc]
atComp Pc Fc = [Pc]

atComp D  P  = [P]
atComp D  Pc = [Pc]
atComp D  D  = [D]
atComp D  Dc = [I, P, Pc, D, Dc, O, Oc, M, Mc, S, Sc, F, Fc]
atComp D  O  = [P, D, O, M, S]
atComp D  Oc = [Pc, D, Oc, Mc, F]
atComp D  M  = [P]
atComp D  Mc = [Pc]
atComp D  S  = [D]
atComp D  Sc = [Pc, D, Oc, Mc, F]
atComp D  F  = [D]
atComp D  Fc = [P, D, O, M, S]

atComp Dc P  = [P, Dc, O, M, Fc]
atComp Dc Pc = [Pc, Dc, Oc, Mc, Sc]
atComp Dc D  = [I, D, Dc, O, Oc, S, Sc, F, Fc]
atComp Dc Dc = [Dc]
atComp Dc O  = [Dc, O, Fc]
atComp Dc Oc = [Dc, Oc, Sc]
atComp Dc M  = [Dc, O, Fc]
atComp Dc Mc = [Dc, Oc, Sc]
atComp Dc S  = [Dc, O, Fc]
atComp Dc Sc = [Dc]
atComp Dc F  = [Dc, Oc, Sc]
atComp Dc Fc = [Dc]

atComp O  P  = [P]
atComp O  Pc = [Pc, Dc, Oc, Mc, Sc]
atComp O  D  = [D, O, S]
atComp O  Dc = [P, Dc, O, M, Fc]
atComp O  O  = [P, O, M]
atComp O  Oc = [I, D, Dc, O, Oc, S, Sc, F, Fc]
atComp O  M  = [P]
atComp O  Mc = [Dc, Oc, Sc]
atComp O  S  = [O]
atComp O  Sc = [Dc, O, Fc]
atComp O  F  = [D, O, S]
atComp O  Fc = [P, O, M]

atComp Oc P  = [P, Dc, O, M, Fc]
atComp Oc Pc = [Pc]
atComp Oc D  = [D, Oc, F]
atComp Oc Dc = [Pc, Dc, Oc, Mc, Sc]
atComp Oc O  = [I, D, Dc, O, Oc, S, Sc, F, Fc]
atComp Oc Oc = [Pc, Oc, Mc]
atComp Oc M  = [Dc, O, Fc]
atComp Oc Mc = [Pc]
atComp Oc S  = [D, Oc, F]
atComp Oc Sc = [Pc, Oc, Mc]
atComp Oc F  = [Oc]
atComp Oc Fc = [Dc, Oc, Sc]

atComp M  P  = [P]
atComp M  Pc = [Pc, Dc, Oc, Mc, Sc]
atComp M  D  = [D, O, S]
atComp M  Dc = [P]
atComp M  O  = [P]
atComp M  Oc = [D, O, S]
atComp M  M  = [P]
atComp M  Mc = [I, F, Fc]
atComp M  S  = [M]
atComp M  Sc = [M]
atComp M  F  = [D, O, S]
atComp M  Fc = [P]

atComp Mc P  = [P, Dc, O, M, Fc]
atComp Mc Pc = [Pc]
atComp Mc D  = [D, Oc, F]
atComp Mc Dc = [Pc]
atComp Mc O  = [D, Oc, F]
atComp Mc Oc = [Pc]
atComp Mc M  = [I, S, Sc]
atComp Mc Mc = [Pc]
atComp Mc S  = [D, Oc, F]
atComp Mc Sc = [Pc]
atComp Mc F  = [Mc]
atComp Mc Fc = [Mc]

atComp S  P  = [P]
atComp S  Pc = [Pc]
atComp S  D  = [D]
atComp S  Dc = [P, Dc, O, M, Fc]
atComp S  O  = [P, O, M]
atComp S  Oc = [D, Oc, F]
atComp S  M  = [P]
atComp S  Mc = [Mc]
atComp S  S  = [S]
atComp S  Sc = [I, S, Sc]
atComp S  F  = [D]
atComp S  Fc = [P, O, M]

atComp Sc P  = [P, Dc, O, M, Fc]
atComp Sc Pc = [Pc]
atComp Sc D  = [D, Oc, F]
atComp Sc Dc = [Dc]
atComp Sc O  = [Dc, O, Fc]
atComp Sc Oc = [Oc]
atComp Sc M  = [Dc, O, Fc]
atComp Sc Mc = [Mc]
atComp Sc S  = [I, S, Sc]
atComp Sc Sc = [Sc]
atComp Sc F  = [Oc]
atComp Sc Fc = [Dc]

atComp F  P  = [P]
atComp F  Pc = [Pc]
atComp F  D  = [D]
atComp F  Dc = [Pc, Dc, Oc, Mc, Sc]
atComp F  O  = [D, O, S]
atComp F  Oc = [Pc, Oc, Mc]
atComp F  M  = [M]
atComp F  Mc = [Pc]
atComp F  S  = [D]
atComp F  Sc = [Pc, Oc, Mc]
atComp F  F  = [F]
atComp F  Fc = [I, F, Fc]

atComp Fc P  = [P]
atComp Fc Pc = [Pc, Dc, Oc, Mc, Sc]
atComp Fc D  = [D, O, S]
atComp Fc Dc = [Dc]
atComp Fc O  = [O]
atComp Fc Oc = [Dc, Oc, Sc]
atComp Fc M  = [M]
atComp Fc Mc = [Dc, Oc, Sc]
atComp Fc S  = [O]
atComp Fc Sc = [Dc]
atComp Fc F  = [I, F, Fc]
atComp Fc Fc = [Fc]
```

3.5 Compass Algebras

Now we switch from spatial reasoning of being contained, touching, etc. to methods of reasoning on directions. This starts with simple compass rose directions. Then an indication is given how this may be refined.

3.5.1 Compass

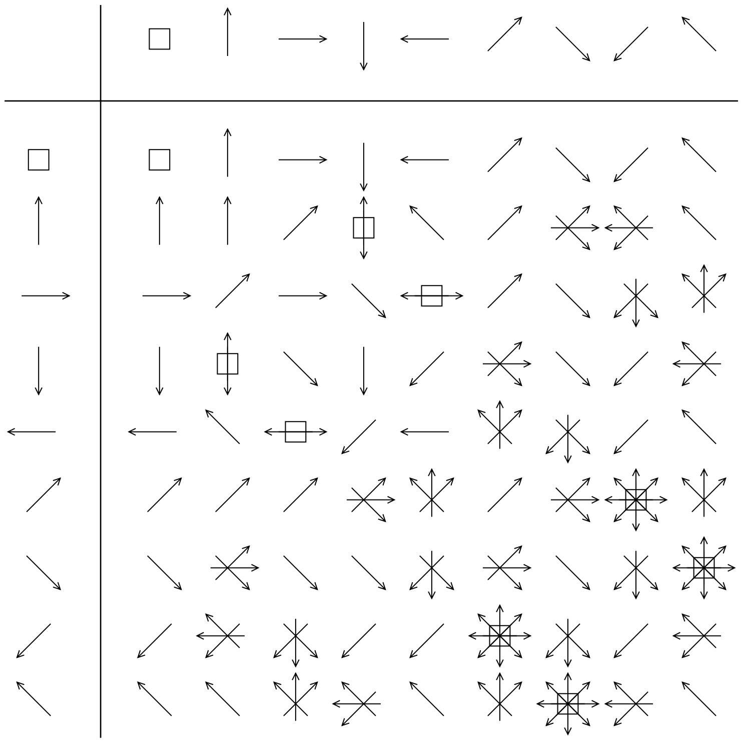
Now we recall the so-called compass algebra. It may be understood as being generated from compass-like qualifications represented in atoms as follows

identity	by	`I`	east	by	`E`	west	by	`W`
			north	by	`N`	south	by	`S`
			northeasterly	by	`NE`	southwesterly	by	`SW`
			northwesterly	by	`NW`	southeasterly	by	`SE`

More details may be found in [VK88, VKvB89, Mad94].

```
module Compass(aCat_Compass,aAll_Compass,ra_Compass,writeAtComp_Compass) where

import RelAlg
import Atomset
import Draw
import FiniteMaps

data Atomset = I | N | E | S | W | NE | SE | SW | NW deriving (Eq, Ord, Show)
atoms = [I, N, E, S, W, NE, SE, SW, NW]

aCat_Compass :: ACat () Atomset
aCat_Compass = ac where
 ac = ACat
  {acat_isObj   = const True
  ,acat_isAtom  = (\ _ _ _ -> True)
  ,acat_objects = [()]
  ,acat_atomset = (\ _ _ -> atoms)
  ,acat_idmor   = acat_idmor_defaultM ac
  ,acat_comp    = (\ _ _ _ -> atComp)
  }

aAll_Compass :: AAll () Atomset
aAll_Compass = AAll
  {aall_acat = aCat_Compass
  ,aall_converse = (\ _ _ -> conv)
  }

ra_Compass :: RA () (SetMor () Atomset)
ra_Compass = atomsetRA aAll_Compass

conv :: Atomset -> Atomset
conv I  = I
conv N  = S
conv E  = W
conv S  = N
conv W  = E
conv NE = SW
conv SE = NW
conv SW = NE
conv NW = SE
```

The explicit composition table is contained in the distribution; here is a nicer presentation:



For producing this, we first have to assign every non-identity atom a direction:

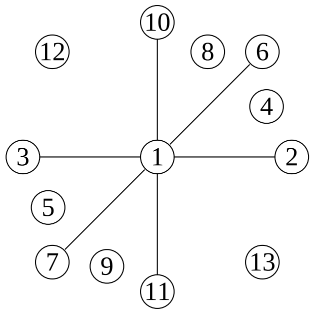
```
atDir N  = 90
atDir E  = 0
atDir S  = 270
atDir W  = 180
atDir NE = 45
atDir SE = 315
atDir SW = 225
atDir NW = 135
atDir I  = error "atDir I"
```

The drawing is then defined by using the tools in the drawing module of Sect. A.4:

```
arrow = "-5 0 moveto 9 0 lineto 7 -1 moveto 9 0 lineto 7 1 lineto "
writeAtComp_Compass =
  let base = graphicCenterFrameScale 10 10 ++ "0.3 setlinewidth\n"
      atPS I = "newpath -3 -3 moveto -3  3 lineto\n" ++
               "         3  3 lineto  3 -3 lineto closepath stroke\n"
      atPS d = "gsave " ++ show (atDir d) ++ " rotate newpath " ++ arrow
               ++ "stroke grestore\n"
      atLout = Limit (MM 9) (MM 9) Empty
      lout = loutPSAtComp'  base atPS atLout 170
                           (MM 7) (MM 7)
                           None NoLength None NoLength
                           aCat_Compass () () ()
  in mkLoutPic "Compass_atComp" lout
```

3.5.2 Refined Compass Algebra

The previous compass algebra may be refined. Here, just one step is added, indicating how this may be done further. We consider vectors in the plain and partition these vectors into 13 subsets. Subset 1 contains just the zero vector. Subsets $2, 3, 6, 7, 10, 11$ contain precisely the nonvanishing vectors of the indicated direction, while subsets $4, 5, 8, 9, 12, 13$ contain all the vectors from the origin that end properly inside the respective region.



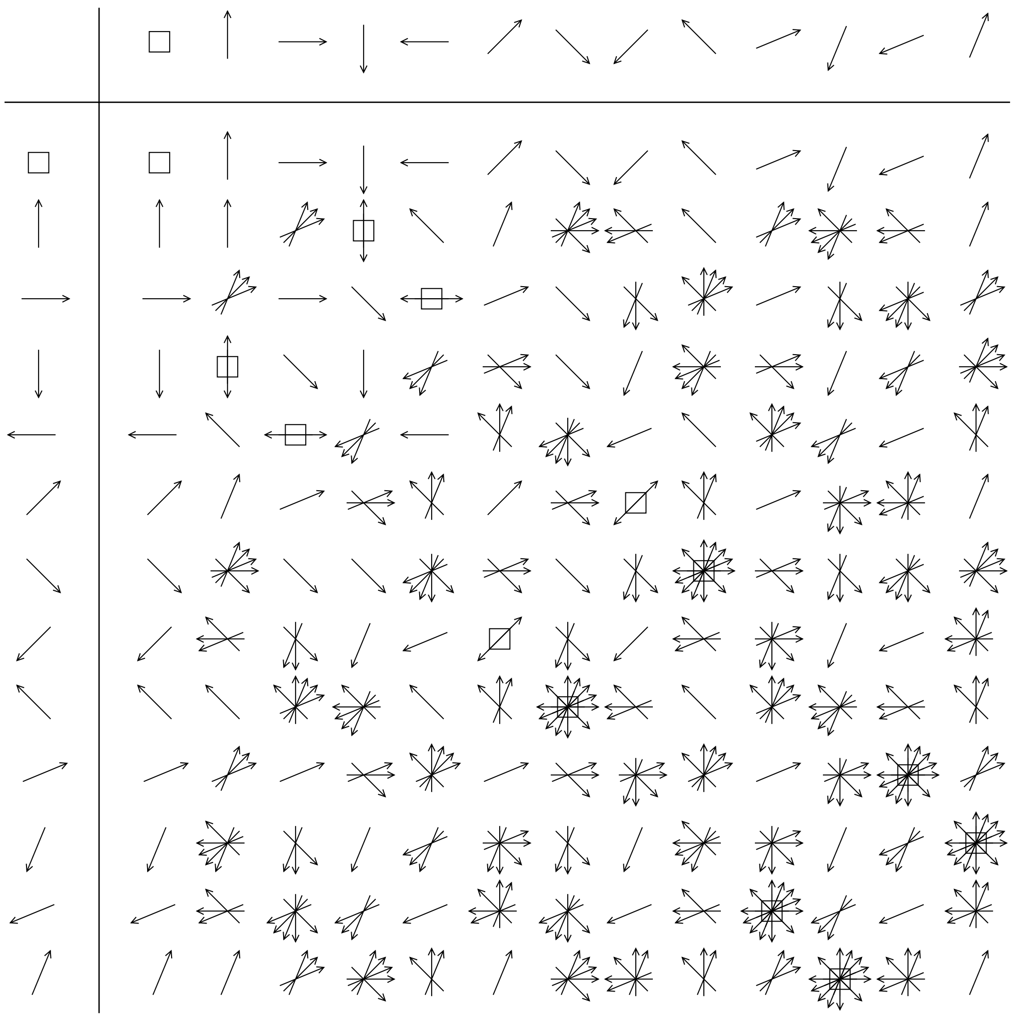
Vector orientation in the compass algebra

It may be understood as being generated from compass-like qualifications represented in atoms as follows:

```
module Compass3(aCat_Compass3,aAll_Compass3,ra_Compass3,writeAtComp_Compass3)
                                                                        where
import RelAlg
import Atomset
import Draw
import FiniteMaps

data Atomset = I | N | E | S | W | NE | SE | SW | NW | ENE | SSW | WSW | NNE
                     deriving (Eq, Ord, Show)
atoms = [I, N, E, S, W, NE, SE, SW, NW, ENE, SSW, WSW, NNE]

aCat_Compass3 :: ACat () Atomset
aCat_Compass3 = ac where
 ac = ACat
  {acat_isObj    = const True
  ,acat_isAtom   = (\ _ _ _ -> True)
  ,acat_objects  = [()]
  ,acat_atomset  = (\ _ _ -> atoms)
  ,acat_idmor    = acat_idmor_defaultM ac
  ,acat_comp     = (\ _ _ _ -> atComp)
  }

aAll_Compass3 :: AAll () Atomset
aAll_Compass3 = AAll
  {aall_acat = aCat_Compass3
  ,aall_converse = (\ _ _ -> conv)
  }
```

```
ra_Compass3 :: RA () (SetMor () Atomset)
ra_Compass3 = atomsetRA aAll_Compass3

conv :: Atomset -> Atomset
conv I  = I
conv N  = S
conv E  = W
conv S  = N
conv W  = E
conv NE  = SW
conv SE  = NW
conv SW  = NE
conv NW  = SE
conv ENE  = WSW
conv SSW  = NNE
conv WSW  = ENE
conv NNE  = SSW
```

As in the last example, we only show a graphical presentation of the composition table:



3.6 Non-Uniform Relation Algebras

One of the unexpected properties of relation algebras we are going to provide examples for is non-uniformity. It may happen that the composition of two universal relations results in a non-universal relation.

```
module NonUniform(aCat_NUW,aAll_NUW,ra_NUW
                 ,nuRA_0,shows_nuRA_0,nuRA_1,nuRA_1',shows_nuRA_1') where

import RelAlg
import Atomset
import A2
import Matrix

import Properties
import Product
import SubAlg

import FiniteMaps
import Sets
```

A simple example of a non-uniform relation algebra is the following:

```
nuRA_0 = raMat raB [[],[()]]
```

This can be checked with:

```
                 performAll ded_uniform_TEST (ra_ded nuRA_0)
```

A direct definition of this can be displayed by calling the following function, evaluating e.g. "`putStr $ shows_nuRA_0 ""`":

```
shows_nuRA_0 = let
    showsObj = (\i s -> "Obj" ++ i ++ s) . show . length . unVec;
    showsAtom a = ("At1"++);
    da = ra_distrAll nuRA_0
 in showsARA' "NU" id showsObj showsAtom (distrAll_aall da)
```

This example, however, includes a trivial homset (i.e., with $\perp\!\!\!\perp = \top\!\!\!\top$).

An example without trivial homsets may, for example be found in Michael Winter's PhD thesis [Win98, p. 24]. It is a heterogeneous relation algebra *with non-trivial homsets* with two objects:

```
data Obj = A | B   deriving (Eq, Ord, Show)
objseq = [A, B]
```

Composition is designed so as to yield
`comp (A, B, [At1]) (B, A, [At1]) = (A, A,[At1])`,
which is properly contained in `(A, A, [At1, At2])`, thus providing an example for

$$\mathbb{T}_{\mathcal{A},\mathcal{B}} ; \mathbb{T}_{\mathcal{B},\mathcal{A}} \neq \mathbb{T}_{\mathcal{A},\mathcal{A}} \ .$$

This relation algebra actually is a sub-algebra of the product relation algebra of `nuRA_0` of the non-uniform relation algebra mentioned in 1.3.3 (that algebra features a trivial homset) with itself, i.e., of the relation algebra

```
nuRA_1 = raProd nuRA_0 nuRA_0
```

This sub-algebra eschews the object `(Vec [],Vec [])` which has a trivial automorphism lattice, but still exhibits non-uniformity; as sub-algebra it is defined in the following way:

```
nuRA_1' = let
   a' = (vec [()],vec [()])
   b' = (vec [()],vec [])
   sub = SubCat (listToSet [a',b']) zeroFM
 in sub_ra (cat_homset_close (ra_cat nuRA_1) sub) nuRA_1
```

Invoking the following (exported) function, e.g. with "`putStr $ showsW1 ""`", prints a definition that is equivalent to the manually generated definition below:

```
shows_nuRA_1' = let
   so (x,y) = case unVec y of [] -> ("B" ++); _ -> ("A" ++)
   sa (x,y) = let ([l],_,_) = unMatMor x
              in case l of [True] -> ("At1" ++); _ -> ("At2" ++)
 in showsARA' "NuRA1a" id so sa (distrAll_aall (ra_distrAll nuRA_1'))
```

Here now the original definition:

```
atoms :: Obj -> Obj -> [A2]
atoms A A = [At1, At2]
atoms _ _ = [At1]

aCat_NUW :: ACat Obj A2
aCat_NUW = ac where
 ac = ACat
  {acat_isObj   = const True
  ,acat_isAtom  = (\ s t a -> a `elem` atoms s t)
  ,acat_objects = objseq
  ,acat_atomset = atoms
  ,acat_idmor   = acat_idmor_defaultM ac
  ,acat_comp    = atComp
  }
```

```
aAll_NUW :: AAll Obj A2
aAll_NUW = AAll
  {aall_acat = aCat_NUW
  ,aall_converse = transpTab
  }

ra_NUW :: RA Obj (SetMor Obj A2)
ra_NUW = atomsetRA aAll_NUW

transpTab :: Obj -> Obj -> A2 -> A2
transpTab _ _ x  = x

atComp :: Obj -> Obj -> Obj -> A2 -> A2 -> [A2]

atComp A A A At1 At1 = [At1]
atComp A A A At2 At1 = []
atComp A A A At1 At2 = []
atComp A A A At2 At2 = [At2]

atComp A A B At1 At1 = [At1]
atComp A A B At2 At1 = []

atComp B A A At1 At1 = [At1]
atComp B A A At1 At2 = []

atComp B A B At1 At1 = [At1]

atComp A B A At1 At1 = [At1]

atComp _ _ _ At1 At1 = [At1]
```

The object `A` is a unit.

3.7 LRNnoc

The relation algebra considered here is intended to show how truly parallel nonstrict behaviour may be modelled by relation algebras, too. We already have accepted to consider the one-element set $\mathbb{B}^0$ as a boolean algebra. Having done this, we will find out that here the one-element morphism set `Mor(L, R)` with an element that should be called *noc* (for: not connected) is an adequate modelling of the corresponding property in net situations.

```
module LRNnoc(aCat_LRNnoc,aAll_LRNnoc,ra_LRNnoc) where

import RelAlg
import Atomset
import A2
import Matrix
import FiniteMaps
import Sets

data Obj = L | R | N  deriving (Eq, Ord, Show)
objseq = [L, R, N]
```

There is one atom to indicate relationship on `L` or on `R`. In the same way, there are two atoms for relationships on `N`, which is intended to model the parallel product of `L` and `R`. There are, however, no atoms provided for relations between `L` and `R`.

```
atoms :: Obj -> Obj -> [A2]
atoms N N = [At1, At2]
atoms R L = []
atoms L R = []
atoms _ _ = [At1]



aCat_LRNnoc :: ACat Obj A2
aCat_LRNnoc = ac where
 ac = ACat
  {acat_isObj   = const True
  ,acat_isAtom  = (\ s t a -> a `elem` atoms s t)
  ,acat_objects = objseq
  ,acat_atomset = atoms
  ,acat_idmor   = acat_idmor_defaultM ac
  ,acat_comp    = atComp
  }

aAll_LRNnoc :: AAll Obj A2
aAll_LRNnoc = AAll
  {aall_acat = aCat_LRNnoc
  ,aall_converse = const $ const id
  }

ra_LRNnoc :: RA Obj (SetMor Obj A2)
ra_LRNnoc = atomsetRA aAll_LRNnoc


atComp :: Obj -> Obj -> Obj -> A2 -> A2 -> [A2]

atComp L _ R  _   _  = []
atComp R _ L  _   _  = []

atComp N R N At1 At1 = [At2]
atComp N N N At2 At2 = [At2]

atComp N N R At2 At1 = [At1]
atComp R N N At1 At2 = [At1]
atComp N N N At2 At1 = []
atComp N N N At1 At2 = []
atComp N N L At2 At1 = []
atComp N N R At1 At1 = []
atComp L N N At1 At2 = []
atComp R N N At1 At1 = []

atComp _ _ _ At1 At1 = [At1]
```

One will observe that there is no possibility to propagate anything from `L` to `R`, not even via `N`. So this models work on two components of a pair, that is executed completely independently. Work on the first component done by `L` cannot influence the second component of the pair.

From the evaluation of the expression

```
printAllTestResults $ all_mapTest $ ra_all ra_LRNnoc
```

we see that the only two non-identical mappings in this relation algebra are

`SetMor ({At1},L,N)` and `SetMor ({At1},R,N)`.

Conclusion and Outlook

In order to enable exploration of non-standard relation algebras, we have first of all defined a layered set of *interfaces* that directly allows access to the mathematical properties of category, allegory, and relation algebra data structures in Haskell. In particular, we have provided extensive tests for well-definedness — building these tests into the data structures themselves would have reduced efficiency in an unacceptable manner.

We then proceeded to present a combinator library for *constructing* such structures using standard mathematical recipes.

Finally we have put together a small collection of *non-standard relation algebras* that illustrate a few different ways in which non-standard relation algebras can fail "conventional relational intuition".

The first serious application we have in mind for our toolkit is the exploration of unsharp models for concurrent computations; here we would need the capability of our toolkit to define and explore *non-standard* relation algebras.

But even for dealing with *standard* relation algebras, i.e., with relation algebras of sets and concrete relations between them, the interfaces provided by our toolkit provide useful tools for exploration and programming. There is, of course, the drawback that our naïve Haskell-list implementation of Boolean matrices is not particularly efficient; here a more direct implementation should be connected to our interfaces. In particular it would be attractive to use the foreign-function interface of current Haskell systems to connect with the efficient implementation of concrete relations that is the kernel of RelView [BBS97]. In this way we would combine the speed of their current BDD implementation of Boolean matrices with access via our flexible Haskell interfaces, opening up new ways to explore relational programming.

Appendix A

Accessories

In this appendix, we collect further parts of our system. First we present several interface modules that may serve as entries for the whole system. Next we include a Haskell 98 module `Main` which also contains quite a few example test calls as candidates for the global `main` binding.

Furthermore, the prelude extensions used in a few places are listed here, and a module `Draw` providing a preliminary drawing interface.

A.1 Interface Modules

For avoiding that all the modules defined in this report need to be imported explicitely by the user, we here include modules that import convenient chunks of our toolkit and re-export everything. This way, just one or two `import` declarations are needed in user programs.

`RATH` — The Comprehensive Haskell 98 Interface

The module `RATH` re-exports all Haskell 98 material from chapters 1 and 2, and also the drawing toolkit from Sect. A.4:

```
module RATH(module RelAlg
           ,module Properties
           ,module Iterations
           ,module Product
           ,module SubAlg
           ,module Matrix
           ,module Atomset
           ,module Draw
           ) where

import RelAlg
import Properties
import Iterations
import Product
import SubAlg
import Matrix
import Atomset

import Draw
```

`RATHexamples` — All Examples

The module `RATHexamples` collects and re-exports all examples from Chapter 3, together with a few small standard relation algebras (`AtomsetExamples`, `P012`, `P0123`) that are not included in this report (they can be generated using the tools of 2.4.8), but are contained in the distribution.

```
module RATHexamples
           (module RATH
           ,module AtomsetExamples
           ,module P012
           ,module P0123
           ,module McKenzie
           ,module Maddux
           ,module NonUniform
           ,module MereoN1
           ,module MereoC1
           ,module MereoG
           ,module Interval
           ,module Compass
           ,module Compass3
           ,module LRNnoc
           ) where

import RATH

import AtomsetExamples
import P012
import P0123
import McKenzie
import Maddux
import NonUniform
import MereoN1
import MereoC1
import MereoG
import Interval
import LRNnoc
import Compass
import Compass3
```

Using the Class Interface

In systems that support multi-parameter type classes and functional dependencies, instead of `RATH`, you may use the extended variant `RATHclasses` that also re-exports the class interfaces of Sect. 1.1 and their instantiations of Sect. 1.4:

```
module RATHclasses(module RATH
                  ,module RelAlgClasses
                  ,module RelAlgInstances) where

import RATH
import RelAlgClasses
import RelAlgInstances
```

Everything Included

For playing with everything this toolkit has to offer, you need a system that supports functional dependencies, which currently essentially means Hugs — just start

“`hugs -98 HugsMain.lhs`”:

```
module HugsMain(module RATHclasses, module RATHexamples) where

import RATHclasses

import RATHexamples
```

A.2 Test Program

```
module Main where

import RATH

import RATHexamples

-- all atomset algebras:
-- P1 P12 P012 P0123 McKenzie Maddux NUW MereoN1 MereoC1 MereoG
--Interval LRNnoc Compass Compass3

main = acat_TESTALL

acat_TESTALL = do
        putStrLn "P1:"
        perform acat_TEST aCat_P1
        putStrLn "P12:"
        perform acat_TEST aCat_P12
        putStrLn "P012:"
        perform acat_TEST aCat_P012
        putStrLn "P0123:"
        perform acat_TEST aCat_P0123
        putStrLn "McKenzie:"
        perform acat_TEST aCat_McKenzie
        putStrLn "Maddux:"
        perform acat_TEST aCat_Maddux
        putStrLn "NUW:"
        perform acat_TEST aCat_NUW
        putStrLn "MereoN1:"
        perform acat_TEST aCat_MereoN1
        putStrLn "MereoC1:"
        perform acat_TEST aCat_MereoC1
        putStrLn "MereoG:"
        perform acat_TEST aCat_MereoG
        putStrLn "Interval:"
        perform acat_TEST aCat_Interval
        putStrLn "LRNnoc:"
        perform acat_TEST aCat_LRNnoc
        putStrLn "Compass:"
        perform acat_TEST aCat_Compass
        putStrLn "Compass:"
        perform acat_TEST aCat_Compass3


aall_TESTALL = do
        putStrLn "P1:"
        perform aall_TEST aAll_P1
        putStrLn "P12:"
        perform aall_TEST aAll_P12
        putStrLn "P012:"
        perform aall_TEST aAll_P012
        putStrLn "P0123:"
        perform aall_TEST aAll_P0123
        putStrLn "McKenzie:"
        perform aall_TEST aAll_McKenzie
        putStrLn "Maddux:"
        perform aall_TEST aAll_Maddux
        putStrLn "NUW:"
        perform aall_TEST aAll_NUW
        putStrLn "MereoN1:"
        perform aall_TEST aAll_MereoN1
        putStrLn "MereoC1:"
        perform aall_TEST aAll_MereoC1
        putStrLn "MereoG:"
        perform aall_TEST aAll_MereoG
        putStrLn "Interval:"
        perform aall_TEST aAll_Interval
        putStrLn "LRNnoc:"
        perform aall_TEST aAll_LRNnoc
        putStrLn "Compass:"
        perform aall_TEST aAll_Compass

units_ALL = do
        putStrLn ("P1: " ++ show (all_units $ atomsetAll aAll_P1))
        putStrLn ("P12: " ++ show (all_units $ atomsetAll aAll_P12))
        putStrLn ("P012: " ++ show (all_units $ atomsetAll aAll_P012))
        putStrLn ("P0123: " ++ show (all_units $ atomsetAll aAll_P0123))
        putStrLn ("McKenzie: " ++ show (all_units $ atomsetAll aAll_McKenzie))
        putStrLn ("Maddux: " ++ show (all_units $ atomsetAll aAll_Maddux))
        putStrLn ("NUW: " ++ show (all_units $ atomsetAll aAll_NUW))
        putStrLn ("MereoN1: " ++ show (all_units $ atomsetAll aAll_MereoN1))
        putStrLn ("MereoC1: " ++ show (all_units $ atomsetAll aAll_MereoC1))
        putStrLn ("MereoG: " ++ show (all_units $ atomsetAll aAll_MereoG))
        putStrLn ("Interval: " ++ show (all_units $ atomsetAll aAll_Interval))
        putStrLn ("LRNnoc: " ++ show (all_units $ atomsetAll aAll_LRNnoc))
        putStrLn ("Compass: " ++ show (all_units $ atomsetAll aAll_Compass))

aall_Products aall = ded_NonemptyProducts $ atomsetDed aall

atomset_PRODUCTS = do
        putStr (unlines ("P1:" : map show (aall_Products aAll_P1)))
        putStr (unlines ("P12:" : map show (aall_Products aAll_P12)))
        putStr (unlines ("P012:" : map show (aall_Products aAll_P012)))
        putStr (unlines ("P0123:" : map show (aall_Products aAll_P0123)))
        putStr (unlines ("McKenzie:" : map show (aall_Products aAll_McKenzie)))
        putStr (unlines ("Maddux:" : map show (aall_Products aAll_Maddux)))
        putStr (unlines ("NUW:" : map show (aall_Products aAll_NUW)))
        putStr (unlines ("MereoN1:" : map show (aall_Products aAll_MereoN1)))
        putStr (unlines ("MereoC1:" : map show (aall_Products aAll_MereoC1)))
        putStr (unlines ("MereoG:" : map show (aall_Products aAll_MereoG)))
        putStr (unlines ("Interval:" : map show (aall_Products aAll_Interval)))
        putStr (unlines ("LRNnoc:" : map show (aall_Products aAll_LRNnoc)))
        putStr (unlines ("Compass:" : map show (aall_Products aAll_Compass)))

all_RA_TEST_ALL = do
        putStrLn "ra1:"
        perform ra_TEST_ALL (ra1 () ())
        putStrLn "ra2:"
        perform ra_TEST_ALL (ra2 42 "Bottom" "Identity")
        putStrLn "raB:"
        perform ra_TEST_ALL raB
        putStrLn "raN2:"
        perform ra_TEST_ALL (raN () 1)
        putStrLn "raN3:"
        perform ra_TEST_ALL (raN () 2)
        putStrLn "P1:"
        perform ra_TEST_ALL ra_P1
        putStrLn "P12:"
        perform ra_TEST_ALL ra_P12
        putStrLn "McKenzie:"
        perform ra_TEST_ALL ra_McKenzie
        putStrLn "NUW:"
        perform ra_TEST_ALL ra_NUW
        putStrLn "MereoN1:"
        perform ra_TEST_ALL ra_MereoN1
        putStrLn "MereoC1:"
        perform ra_TEST_ALL ra_MereoC1
        putStrLn "MereoG:"
        perform ra_TEST_ALL ra_MereoG
        putStrLn "Interval:"
        perform ra_TEST_ALL ra_Interval
        putStrLn "LRNnoc:"
        perform ra_TEST_ALL ra_LRNnoc
        putStrLn "P012:"
        perform ra_TEST_ALL ra_P012
        putStrLn "Compass:"
        perform ra_TEST_ALL ra_Compass
        putStrLn "Maddux:"
        perform ra_TEST_ALL ra_Maddux
        putStrLn "P0123:"
        perform ra_TEST_ALL ra_P0123

-- main = all_RA_TEST_ALL


-- main = perform cat_TEST $
--        ra_cat ra_P012                                 -- 0.290s
--        catMat distrAllB [[],[()]]                     -- 0.010s
--        catMat distrAllB o012                          -- 0.410s
--        catMat distrAllB o0123                         -- 4h45m56.950s
--        catMat (ra_distrAll ra_P012) [[P0],[P1,P1],[P2]]    -- 6.940s
--        catMat (ra_distrAll ra_P012) [[P0],[P1],[P1,P1],[P2]] -- 8.770s

-- main = perform all_TEST $
--        ra_all ra_P012                                 -- 0.320s
--        allMat distrAllB o012                          -- 0.440s
--        allMat distrAllB o0123                         --
--        allMat (ra_distrAll ra_P012) [[P0],[P1,P1],[P2]]     -- 4.460s
--        allMat (ra_distrAll ra_P012) [[P0],[P1],[P1,P1],[P2]]   -- 4.780s
--        allMat (ra_distrAll ra_P012) [[P0],[P1],[P1,P1],[P2],[P1,P2]] --

--main = test_for_equivalence_0123

inits l = [] : case l of [] -> [];  (x:xs) -> map (x:) (inits xs)

trivs n = inits (replicate n ())

o012 = trivs 2
o0123 = trivs 3
o01234 = trivs 4
```

```
-- main = perform cat_TEST $ atomsetCat $ acatMat acatB o012  -- 0.350s
-- main = perform all_TEST $ atomsetAll $ aallMat aallB o012  -- 0.500s

--main = perform acat_TEST aCat_P012 -- 0.020s
--main = perform aall_TEST aAll_P012 -- 0.000s
--main = perform acat_TEST $ acatMat aCat_P012 [[P0],[P1],[P1,P1],[P2]] -- 0.030s
--main = perform aall_TEST $ aallMat aAll_P012 [[P0],[P1],[P1,P1],[P2]] -- 0.010s

-- main = perform acat_TEST $ acatMat acatB o012      -- 0.000s
-- main = perform aall_TEST $ aallMat aallB o012      -- 0.000s
-- main = perform acat_TEST $ acatMat acatB o0123     -- 0.100s
-- main = perform aall_TEST $ aallMat aallB o0123     -- 0.030s
-- main = perform acat_TEST $ acatMat acatB o01234    -- 0.990s
-- main = perform aall_TEST $ aallMat aallB o01234    -- 0.220s
-- main = perform acat_TEST $ acatMat acatB $ trivs 5 -- 6.610s
-- main = perform aall_TEST $ aallMat aallB $ trivs 5 -- 1.210s
-- main = perform acat_TEST $ acatMat acatB $ trivs 6 -- 34.030s
-- main = perform aall_TEST $ aallMat aallB $ trivs 6 -- 5.150s

-- main = perform cat_TEST $ atomsetCat $
--        acatMat aCat_P012 [[P0],[P1],[P1,P1],[P2]] -- 5.490s

-- main = perform all_TEST $ atomsetAll $
--        aallMat aAll_P012 [[P0],[P1],[P1,P1],[P2]] -- 3.640s

-- main = perform ra_TEST_ALL $ atomsetRA $
--        aallMat aAll_P012 [[P0],[P1],[P1,P1],[P2]] -- 8.670s

-- main = perform ra_TEST_ALL $ ra_McKenzie -- 0.740s

-- main = perform acat_TEST aCat_Maddux   -- 0.200s
-- main = perform aall_TEST aAll_Maddux   -- 0.030s
-- main = perform ra_TEST_ALL $ ra_Maddux -- 19.600s

matObjs_Maddux = let objseq = acat_objects aCat_Maddux in
                 do a <- objseq
                    [a] : do b <- objseq
                             [[a,b]]

matDed_Maddux = atomsetDed $ aallMat aAll_Maddux matObjs_Maddux

--matDed_Maddux_NEProds = ded_NonemptyProducts matDed_Maddux

--main = putStr $ unlines $ map show $ ded_NonemptyProducts matDed_Maddux
```

A.3 Prelude Extensions

```
module ExtPrel where


prodF f g (x,y) = (f x, g y)

prodFF f g (x1,x2) (y1,y2)  = (f x1 y1, g x2 y2)

cprodFF h f g (x1,x2) (y1,y2)  = h (f x1 y1, g x2 y2)

prodFFF f g (x1,x2) (y1,y2) (z1,z2)  = (f x1 y1 z1, g x2 y2 z2)

cprodFFF h f g (x1,x2) (y1,y2) (z1,z2)  = h (f x1 y1 z1, g x2 y2 z2)

prodPP2 p (x1,y1) (x2,y2) = p y1 y2


tupd_3_1 f (x, y, z) = (f x, y, z)


cTrue x = True
cFalse x = False
ccTrue x y = True
ccFalse x y = False


listProd (as,bs) = [(a,b) | a <- as, b <- bs]


power :: [a] -> [[a]]
power l = power' id l []
 where
  power' f [] = ((f []):)
  power' f (x:xs) = power' f xs . power' (f . (x:)) xs


type FctS a b = [(a,b)] -> [(a,b)]
type FctsS a b = FctS a b -> [[(a,b)]] -> [[(a,b)]]

totFct :: [a] -> [b] -> [[(a,b)]]
-- totFct (dom :: [a]) (ran :: [b]) =
totFct dom ran =
 foldr h (\ f -> ((f []) :)) dom id []
  where
    -- h :: a -> FctsS a b -> FctsS a b
    h x mkfs = foldr k (const id) ran
      where -- k :: b -> FctsS a b -> FctsS a b
            k y mkfs' f = mkfs (f . ((x,y):)) . mkfs' f
```

```
pairAnd = uncurry (&&)


insertSet :: Ord a => a -> [a] -> [a]
insertSet x []         =  [x]
insertSet x ys@(y:ys') =  case compare x y of
                                GT -> y : insertSet x ys'
                                EQ -> ys
                                _  -> x : ys

listEqAsSet :: Ord a => [a] -> [a] -> Bool
listEqAsSet xs ys = foldr insertSet [] xs == foldr insertSet [] ys


listShowsSep shows c = h
  where h [] = id
        h [x] = shows x
        h (x:xs) = shows x . (c :) . h xs

listShows shows xs = ('[' :) . listShowsSep shows ',' xs . (']' :)


foldl'           :: (a -> b -> a) -> a -> [b] -> a
foldl' f a []     = a
foldl' f a (x:xs) = (foldl' f $! f a x) xs

length' :: Integral i => [a] -> i
length' = foldl' (\ n _ -> n + 1) 0

rcurry :: ((a,b) -> c) -> b -> a -> c
rcurry f = flip (curry f)


unfold :: (a -> (b,a)) -> a -> [b]
unfold f x = let (r,y) = f x in r : unfold f y


newtype STFun s a = STFun (s -> (s,a))
applySTFun (STFun f) = f

instance Functor (STFun s) where
  fmap f (STFun g) = STFun (\s -> let (s', a) = g s
                                  in  (s', f a))

instance Monad (STFun s) where
  return x = STFun (\s -> (s,x))
  (STFun f) >>= g  = STFun (\s -> let (s',a) = f s
                                      STFun g' = g a
                                  in g' s')


untilFix f x = let x' = f x in if x' == x then x else untilFix f x'
```

A.4 Drawing

Drawings for concrete relations can for example choose a Boolean matrix representation, or a (directed) graph representation — both possibilities are implemented in the RelView system [BBS97].

For abstract categories or allegories, there is no similarly general approach. For relation algebras, we may use the atoms to arrive at a general representation of their morphisms. Similarly, as soon as morphisms of a coefficient allegory have a graphical representation, we may use that to build matrix representations.

In this module we provide a very simple drawing interface for some of the constructions of Chapter 2. We target PostScript generation, delegating some of the tasks to the document formatting system Lout [Kin95].

```
module Draw where

import System

import RelAlg
import Atomset
import Matrix

data Length = NoLength | CM Double | Pt Double   | MM Double
                                   | Inch Double | FontSize Double
lengthToLoutStr NoLength = ""
lengthToLoutStr (CM l) = shows l "c"
lengthToLoutStr (Pt l) = shows l "p"
lengthToLoutStr (MM l) = shows (l/10.0) "c"
lengthToLoutStr (Inch l) = shows l "i"
lengthToLoutStr (FontSize l) = shows l "f"

data GapMode = None | Edge
instance Show GapMode where
  showsPrec _ None = id
  showsPrec _ Edge = ('e':)

type PostScript = String
```

Because of its flexibility and powerful alignment operations, we use a fragment of the graphical object model of Lout [Kin95]:

```
data Lout = PS PostScript Lout
          | HCat Bool GapMode Length [Lout]
          | VCat Bool GapMode Length [Lout]
          | Scale (Maybe (Either Double (Double, Double))) Lout
          | Wrap Length Length Lout
          | Box Length Length Lout
          | Limit Length Length Lout
```

```
        | Empty

ps w h ps = PS ps (Limit w h Empty)
```

Since we also use the implementation of Lout to generate images, here we define an appropriate instance of `Show` that produces legal Lout code:

```
instance Show Lout where
  showsPrec _ (Limit w h l) =
    (case w of NoLength -> id
               _ -> ((lengthToLoutStr h ++ " @Wide ") ++)) .
    (case h of NoLength -> id
               _ -> ((lengthToLoutStr h ++ " @High ") ++)) . shows l
  showsPrec _ Empty = ("{}\n"++)
  showsPrec _ (PS ps l) = ("{\n" ++) . (ps++) . ("\n}\n@Graphic { "++) .
      shows l . ("}\n" ++)
  showsPrec _ (Box lw m l) = ("@Box " ++) .
    (case lw of NoLength -> id
                _ -> (("linewidth {" ++ lengthToLoutStr lw ++ "} ") ++)) .
    (case m of NoLength -> id
               _ -> (("margin {" ++ lengthToLoutStr m ++ "} ") ++)) .
    ("{\n" ++) . shows l . ("}\n" ++)
  showsPrec _ (Wrap h v l) =
    ((lengthToLoutStr v ++ " @High {} // {\n") ++) .
    ((lengthToLoutStr h ++ " @Wide {} || {\n") ++) .
    shows l .
    (("} || " ++ lengthToLoutStr h ++ " @Wide {}\n") ++) .
    (("} // " ++ lengthToLoutStr v ++ " @High {}\n") ++)
  showsPrec _ (Scale s l) =
     ('{':) . scale . (" @Scale " ++) . shows l . ("}\n"++)
   where scale = case s of
           Nothing -> id
           Just (Left f) -> shows f
           Just (Right (h,v)) -> ('{':) . shows h . (',':) . shows v . ('}':)
  showsPrec _ (HCat b m w ls) = case ls of
     [] -> ("{}" ++)
     [l] -> shows l
     _ -> ("{\n" ++) . fold1 hcat shows id ls . ("}\n"++)
   where hcat x y = shows x . ((if b then "|" else "||") ++) .
                    ((lengthToLoutStr w) ++) . shows m . ('\n' : ) . y

  showsPrec _ (VCat b m w ls) = case ls of
     [] -> ("{}" ++)
     [l] -> shows l
     _ -> ("{\n" ++) . fold1 vcat shows id ls . ("}\n"++)
   where vcat x y = shows x . ((if b then "/" else "//") ++) .
                    ((lengthToLoutStr w) ++) . shows m . ('\n' : ) . y
```

```
fold1 f g e [] = e
fold1 f g e [x] = g x
fold1 f g e (x : xs) = f x (fold1 f g e xs)
```

The most important use of the alignment operations is for producing matrix drawings:

```
loutMatrix :: GapMode -> Length -> GapMode -> Length -> [[Lout]] -> Lout
loutMatrix gh dh gv dv m = VCat True gv dv $ map (HCat True gh dh) m

psMatrix :: GapMode -> Length -> GapMode -> Length -> [[PostScript]] -> Lout
psMatrix gh dh gv dv m = loutMatrix gh dh gv dv (map (map f) m)
 where f p = ps (MM 4) (MM 4) (p ++ '\n' : graphicFramePath ++ "stroke\n")

graphicFramePath = "0     0     moveto    0     ysize lineto\n" ++
                   "xsize ysize lineto    xsize 0     lineto closepath\n"

graphicFrameScale h v = "xsize " ++ shows h " div\n"
                     ++ "ysize " ++ shows v " div scale\n"
graphicCenterFrameScale h v =
 "xsize 2 div ysize 2 div translate\n" ++
 "xsize " ++ shows h " 2 mul div\n" ++
 "ysize " ++ shows v " 2 mul div scale\n"
```

Since for RelView-like output of Boolean matrices we also need that frame for filling, we additionally provide an abbreviated variant:

```
boolMatElem :: Bool -> Lout
boolMatElem b = ps (MM 4) (MM 4) (
  graphicFramePath
  ++ if b then "gsave 0.7 setgray fill grestore stroke\n" else "stroke\n"
 )

defaultLoutMatrix = loutMatrix None NoLength None NoLength
boolMatLout m = defaultLoutMatrix (map (map boolMatElem) m)

loutMatMor :: Length -> Length ->
              GapMode -> Length -> GapMode -> Length ->
              (obj -> Lout) -> (mor -> Lout) -> MatMor obj mor -> Lout
loutMatMor seph sepv gh dh gv dv objLout morLout mm =
  let (m,s,t) = unMatMor mm
      mat = loutMatrix gh dh gv dv $ map (map morLout) m
      src = VCat True gv dv $ map objLout s
      trg = HCat True gh dh $ map objLout t
  in VCat True None sepv
        [HCat True None seph [Empty,trg], HCat True None seph [src,mat]]
```

For atom set relations, we rely on the verbatim PostScript inclusion features of Basser Lout with the idea that every atom has some PostScript encoding, and the PostScript encodings of all atoms present in a relation are overlaid to produce the presentation of that relation. For examples where this works out nicely see the compass algebras in Sect. 3.5.

At first we define a function that expects separate Lout-producing functions for atoms and lists of atoms:

```
loutAtComp :: Length -> Length ->
              GapMode -> Length -> GapMode -> Length ->
              (atom -> Lout) -> ([atom] -> Lout) ->
              ACat obj atom -> obj -> obj -> obj -> Lout
loutAtComp seph sepv gh dh gv dv atLout morLout ac o1 o2 o3 =
  let atoms1 = acat_atomset ac o1 o2
      atoms2 = acat_atomset ac o2 o3
      m = map
          (\ a1 -> map (\ a2 -> morLout (acat_comp ac o1 o2 o3 a1 a2)) atoms2)
          atoms1
      mat = loutMatrix gh dh gv dv m
      src = VCat True gv dv $ map atLout atoms1
      trg = HCat True gh dh $ map atLout atoms2
  in VCat True None sepv [HCat True None seph [Empty,trg],
                          HCat True None seph [src,mat]]
```

These two functions are now produced together, based on a common PostScript prologue, a function turning individual atoms into PostScript code fragments, and a fixed Lout object to wrap the PostScript around:

```
mkAtMorLout :: PostScript -> (atom -> PostScript) -> Lout ->
               (atom -> Lout, [atom] -> Lout)
mkAtMorLout base atPS atLout =
  (\ at -> PS (base ++ atPS at) atLout
  ,\ mor -> PS (base ++ concatMap atPS mor) atLout
  )

loutPSAtComp :: PostScript -> (atom -> PostScript) -> Lout ->
    Length -> Length -> GapMode -> Length -> GapMode -> Length ->
    ACat obj atom -> obj -> obj -> obj -> Lout
loutPSAtComp base atPS atLout  seph sepv gh dh gv dv ac o1 o2 o3 =
  loutAtComp seph sepv gh dh gv dv atomLout morLout ac o1 o2 o3
   where (atomLout, morLout) = mkAtMorLout base atPS atLout
```

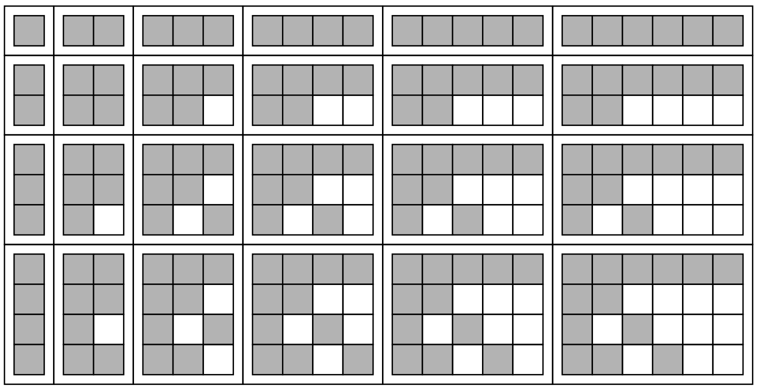
For obtaining separating lines between the composition table proper and its labellings we might revert to Lout's tables instead of employing plain object compositions; here is a simple “hack” that adds the two lines *a posteriori*, and with manually adjusted placement via the argument `corr`:

```
loutPSAtComp' base atPS atLout corr seph sepv gh dh gv dv ac o1 o2 o3 =
  PS
  (unlines
  ["newpath"
  ,"\"/dx\" xsize " ++ show xl ++ " div " ++ show corr ++ " add def"
  ,"\"/dy\" ysize dup " ++ show yl ++ " div " ++ show corr ++ " add sub def"
  ,"dx 0 moveto dx ysize lineto"
  ,"0 dy moveto xsize dy lineto"
  ,"stroke"
  ])
  $
  loutPSAtComp base atPS atLout  seph sepv gh dh gv dv ac o1 o2 o3
 where
  xl = length (acat_atomset ac o1 o2) + 1
  yl = length (acat_atomset ac o2 o3) + 1
```

For testing these capabilities, we play around a little bit:

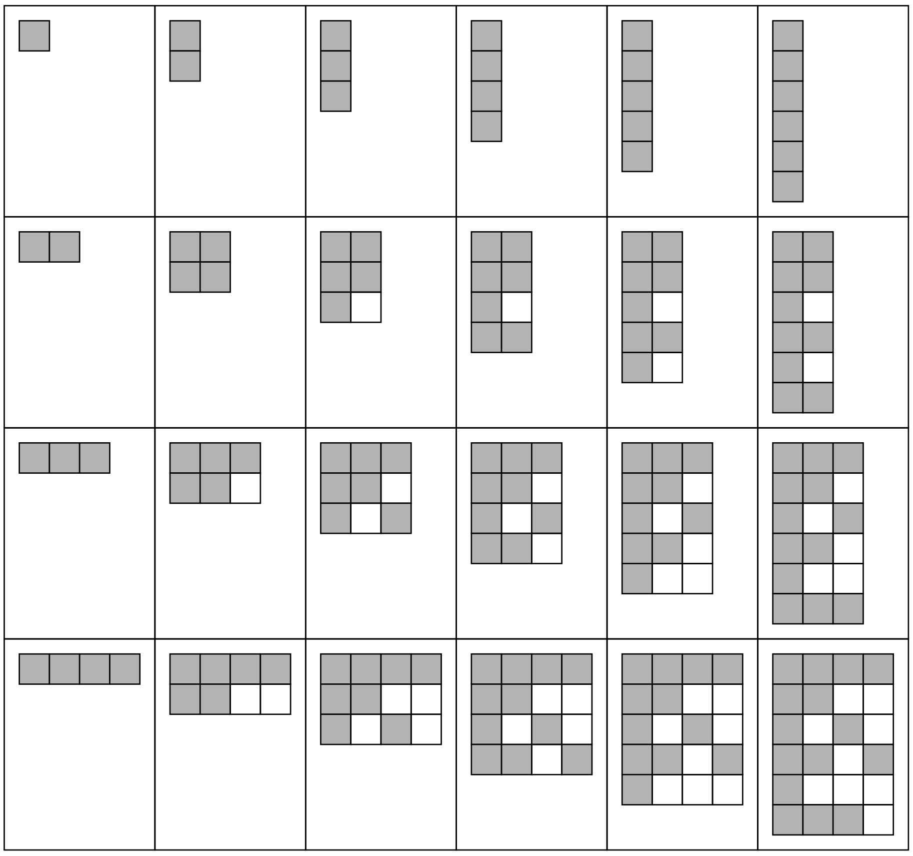
```
mat1 = matX 8 8

matX i j = do r <- [1..i]
              [do c <- [1..j]
                  [r `mod` c == 0 || (r + c >= r * c)]]

mat2 = do i <- [1..4]
          [do j <- [1..6]
              [Box NoLength NoLength (boolMatLout (matX i j))]]

loutDocFile file lout = writeFile file $ unlines
  ["@SysInclude {doc}"
  ,"@Doc @Text @Begin"
  ,show lout
  ,"@End @Text"
  ]

loutPicFile file lout = writeFile file $ unlines
  ["@SysInclude {picture}"
  ,"@Illustration {"
  ,show (Wrap (MM 2) (MM 2) lout)
  ,"}"
  ]

mkLoutPic base lout = let ltfile = base ++ ".lt" in
  do loutPicFile ltfile lout
     system ("lout -EPS -c " ++ base ++ " -o " ++ base ++ ".eps " ++ ltfile)

mk_mat1 = mkLoutPic "mat1" $ boolMatLout mat1
mk_mat2 = mkLoutPic "mat2" $ defaultLoutMatrix mat2
```

This produces the following picture:



For demonstrating the usefulness of Lout's alignment operations we also build a variant with transposed coefficient matrices:

```
mat2a = do i <- [1..4]
           [do j <- [1..6]
               [PS (graphicFramePath ++ "stroke\n")
                   $ Wrap (MM 2) (MM 2) (boolMatLout (matX j i))]]
```



Showing a matrix of matrices

Bibliography

[AK83] James F. Allen and Johannes A. Koomen. Planning using a temporal world model. In *Proc. of the* 8*th* *Internat. Joint Conf. on Artificial Intelligence (IJCAI)*, pages 741–747, Karlsruhe, Germany, August 1983.

[All81] James F. Allen. An interval-based representation of temporal knowledge. In *Proc. of the* 7*th* *Internat. Joint Conf. on Artificial Intelligence, (IJCAI)*, pages 221–226, 1981.

[All83] James F. Allen. Maintaining knowledge about temporal intervals. *Comm. ACM*, 26(11):832–842, November 1983.

[ATBS89] Hilde Abold-Thalmann, Rudolf Berghammer, and Gunther Schmidt. Manipulation of concrete relations: The RELVIEW-System. Technical Report 8905, Universität der Bundeswehr München, Fakultät für Informatik, October 1989.

[BBS97] Ralf Behnke, Rudolf Berghammer, and Peter Schneider. Machine support of relational computations: The Kiel RELVIEW system. Technical Report 9711, Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität Kiel, June 1997.

[BDM97] Richard S. Bird and Oege De Moor. *Algebra of Programming*, volume 100 of *International Series in Computer Science*. Prentice Hall, 1997.

[BG91] L. Biacino and G. Gerla. Connection structures. *Notre Dame J. Formal Logic*, 32:242–247, 1991.

[BH94] Rudolf Berghammer and Claudia Hattensperger. Computer-aided manipulation of relational expressions and formulae using RALF. In Bettina Buth and Rudolf Berghammer, editors, *Systems for Computer-Aided Specification, Development and Verification*, Bericht Nr. 9416, pages 62–78. Universität Kiel, 1994.

[BKS97] Chris Brink, Wolfram Kahl, and Gunther Schmidt, editors. *Relational Methods in Computer Science*. Advances in Computing. Springer, Wien, New York, 1997.

[BSZ86] Rudolf Berghammer, Gunther Schmidt, and Hans Zierer. Symmetric quotients. Technical Report TUM-INFO 8620, Technische Universität München, Fakultät für Informatik, 1986. 18 p.

[BSZ89] Rudolf Berghammer, Gunther Schmidt, and Hans Zierer. Symmetric quotients and domain constructions. *Inform. Process. Lett.*, 33(3):163–168, 1989.

[Car82] Rodrigo Cardoso. *Untersuchung paralleler Programme mit relationenalgebraischen Methoden.* Diplomarbeit under supervision of gunther schmidt, TU München, 1982.

[Coh96] A. G. Cohn. Calculi for qualitative reasoning. In *Artificial Intelligence and Symbolic Mathematical Computation*, volume 1138 of *LNCS*, pages 124–143. Springer, 1996.

[Des99] Jules Desharnais. Monomorphic characterization of n-ary direct products. *Information Sciences*, 119(3–4):275–288, December 1999.

[DM50] Augustus De Morgan. On the symbols of logic, the theory of the syllogism, and in particular of the copula, and the application of the theory of probabilities to some questions in the theory of evidence. *Trans. of the Cambridge Philosophical Society*, 9:79–127, 1850. Reprinted in [DM66].

[DM60] Augustus De Morgan. On the Syllogism: IV; and on the Logic of Relations. *Trans. of the Cambridge Philosophical Society*, 10:331–358, 1860. (dated 12 November 1859) Reprinted in [DM66].

[DM66] Augustus De Morgan. *On the Syllogism, and Other Logical Writings.* Yale Univ. Press, New Haven, 1966.

[DSW99] Ivo Düntsch, Gunther Schmidt, and Michael Winter. A necessary relation algebra for mereotopology. *Studia Logica*, 1999. in print.

[DWM98] Ivo Düntsch, Hui Wang, and Steve McKloskey. Relation algebras in spatial reasoning. In Ewa Orlowska, editor, *Relational Methods in Logic, Algebra and Computer Science, 4th International Seminar RelMiCS, Warsaw, Poland, 14–20 September 1998, Extended Abstracts*, pages 63–68. Stefan Banach International Mathematical Center, Warsaw, 1998.

[FK98] Hitoshi Furusawa and Wolfram Kahl. A study on symmetric quotients. Technical Report 1998-06, Fakultät für Informatik, Universität der Bundeswehr München, December 1998.

[FS90] Peter J. Freyd and Andre Scedrov. *Categories, Allegories*, volume 39 of *North-Holland Mathematical Library.* North-Holland, Amsterdam, 1990.

[Hat97] Claudia Hattensperger. *Rechnergestütztes Beweisen in heterogenen Relationenalgebren.* Dissertationsverlag NG Kopierladen, München, December 1997. ISBN 3-928536-99-0; zugl. Dissertation an der Universität der Bundeswehr München, Fakultät für Informatik.

[HBS94] Claudia Hattensperger, Rudolf Berghammer, and Gunther Schmidt. RALF — A relation-algebraic formula manipulation system and proof checker. Notes to a system demonstration. In Nivat et al. [NRRS94], pages 405–406.

[HPJW$^+$92] Paul Hudak, Simon L. Peyton Jones, Philip Wadler, et al. Report on the programming language Haskell, a non-strict purely functional language, version 1.2. *ACM SIGPLAN Notices*, 27(5), 1992. See also URL: http://haskell.org/.

[Jip92] Peter Jipsen. *Computer-aided investigations of relation algebras.* PhD thesis, Vanderbilt University, May 1992.

[Jón88] Bjarni Jónsson. Relation algebras and Schröder categories. *Discrete Math.*, 70:27–45, 1988.

[Jon00] Mark P. Jones. Type classes with functional dependencies. In G. Smolka, editor, *ESOP 2000*, volume 1782 of *LNCS*, pages 230–244. Springer, March 2000.

[KH98] Wolfram Kahl and Claudia Hattensperger. Second-order syntax in HOPS and in RALF. In Bettina Buth, Rudolf Berghammer, and Jan Peleska, editors, *Tools for System Development and Verification*, volume 1 of *BISS Monographs*, pages 140–164, Aachen, 1998. Shaker Verlag. ISBN: 3-8265-3806-4.

[Kin95] Jeffrey Howard Kingston. *A User's Guide to the Lout Document Formatting System (Version 3).* Basser Department of Computer Science, University of Sydney, 1995. System available from `ftp.cs.su.oz.au:/jeff/lout`.

[Leś29] S. Leśniewski. Grundzüge eines neuen Systems der Grundlagen der Mathematik. *Fund. Math.*, 14:1–81, 1929.

[Lor54] Paul Lorenzen. Über die Korrespondenzen einer Struktur. *Math. Z.*, 60:61–65, 1954. Zbl. Mat. 55 23.

[Lyn50] Roger C. Lyndon. The representation of relational algebras. *Ann. of Math. (2)*, 51:707–729, 1950.

[Mad94] Roger Duncan Maddux. Relation algebras for reasoning about time and space. In Nivat et al. [NRRS94], pages 27–44.

[Mad95] Roger Duncan Maddux. On the derivation of identities involving projection functions. In Csirmaz, Gabbay, and de Rijke, editors, *Logic Colloquium '92*, pages 145–163, Stanford, January 1995. Center for the Study of Language and Information Publications.

[MB83] J. Malik and T. O. Binford. Reasoning in time and space. In *Proc. of the* 8^{th} *Internat. Joint Conf. on Artificial Intelligence, Karlsruhe, W. Germany, August 1983 (IJCAI)*, pages 343–345, 1983.

[McK70] Ralph Nelson Whitfield McKenzie. The representation of integral relation algebras. *Michigan Math. J.*, 17:279–287, 1970.

[NRRS94] Maurice Nivat, Charles Rattray, Teodore Rus, and Giuseppe Scollo, editors. *Proc. 3rd Internat. Conf. Algebraic Methodology and Software Technology, Enschede, June 21–25*, Workshops in Computing. Springer, 1994.

[OS80] J.P. Olivier and D. Serrato. Catégories de Dedekind. Morphismes dans les catégories de Schröder. *C. R. Acad. Sci. Paris Ser. A-B*, 290:939–941, 1980.

[OS95] J.P. Olivier and D. Serrato. Squares and rectangles in relation categories – three cases: Semilattice, distributive lattice and boolean non-unitary. *Fuzzy Sets and Systems*, 72:167–178, 1995.

[Rig48] Jacques Riguet. Relations binaires, fermetures, correspondances de Galois. *Bull. Soc. Math. France*, 76:114–155, 1948.

[Sch95] Ernst Schröder. *Vorlesungen über die Algebra der Logik (exacte Logik).* Teubner, Leipzig, 1895. Vol. 3, Algebra und Logik der Relative, part I, 2nd edition published by Chelsea, 1966.

[Sch77] Gunther Schmidt. Programme als partielle Graphen. Habil. Thesis, Fachbereich Mathematik der Technischen Univ. München, Bericht 7813, 1977. English as [Sch81a, Sch81b].

[Sch81a] Gunther Schmidt. Programs as partial graphs I: Flow equivalence and correctness. *Theoretical Computer Science*, 15:1–25, 1981.

[Sch81b] Gunther Schmidt. Programs as partial graphs II: Recursion. *Theoretical Computer Science*, 15(2):159–179, 1981.

[SS85a] Gunther Schmidt and Thomas Ströhlein. On kernels of graphs and solutions of games — a synopsis based on relations and fixpoints. *SIAM J. Algebraic Discrete Methods*, 6:54–65, 1985.

[SS85b] Gunther Schmidt and Thomas Ströhlein. Relation algebras — concept of points and representability. *Discrete Math.*, 54:83–92, 1985.

[SS89] Gunther Schmidt and Thomas Ströhlein. *Relationen und Graphen.* Mathematik für Informatiker. Springer, Berlin, 1989. English as [SS93].

[SS93] Gunther Schmidt and Thomas Ströhlein. *Relations and Graphs, Discrete Mathematics for Computer Scientists.* EATCS-Monographs on Theoretical Computer Science. Springer, 1993.

[vB83] Johan F.A.K. van Benthem. *The Logic of Time.* Reidel, Dordrecht, NL, 1983.

[VK88] M. Vilain and H. Kautz. Constraint propagation algorithms for temporal reasoning. In H. E. Shrobe, editor, *Proc. AAAI-86*, pages 377–382. Morgan Kaufmann, 1988.

[VKvB89] M. Vilain, H. Krautz, and P. G. van Beek. Constraint propagation algorithms for temporal reasoning. In Weld and de Kleer, editors, *Readings in Qualitative Reasoning About Physical Systems.* Morgan Kaufmann, 1989. Revised version of [VK88].

[vOG97] David von Oheimb and Thomas F. Gritzner. RALL: Machine-supported proofs for relation algebra. In William McCune, editor, *Conference on Automated Deduction - CADE-14*, LNCS 1249, pages 380–394. Springer-Verlag, Berlin, 1997.

[Win98] Michael Winter. *Strukturtheorie heterogener Relationenalgebren mit Anwendung auf nichtdeterminismus in Programmiersprachen.* PhD thesis, Fakultät für Informatik, Universität der Bundeswehr München, April 1998.

[ZSB86] Hans Zierer, Gunther Schmidt, and Rudolf Berghammer. An interactive graphical manipulation system for higher objects based on relational algebra. In Gottfried Tinhofer and Gunther Schmidt, editors, *WG '86*, volume 246 of *LNCS*, pages 68–81, Bernried, Starnberger See, June 1986. Springer.

Index