Theory Extraction in Relational Data Analysis

Gunther Schmidt[1]

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

1 Introduction

From numerical mathematics we know that a linear equation $Ax = b$ may be solved more efficiently if a reduction of A as $A = \begin{pmatrix} B & O \\ C & D \end{pmatrix}$ is known beforehand. For the task $\begin{pmatrix} B & O \\ C & D \end{pmatrix} \cdot \begin{pmatrix} y \\ z \end{pmatrix} = \begin{pmatrix} c \\ d \end{pmatrix}$, one will solve $By = c$ first and then $Dz = d - Cy$. Having an a priori knowledge of this kind is also an advantage in many other application fields. We here deal with a diversity of techniques to decompose relations according to some criteria and embed these techniques in a common framework. The results of decompositions obtained may be used in decision making, but also as a support for teaching, as they often give visual help.

Our starting point will always be a concretely given relation, i.e., a Boolean matrix. In most cases, we will look for a partition of the set of rows and the set of columns, respectively, that arises from some algebraic condition. From these partitions, a rearranged matrix making these partitions easily visible shall be computed as well as the permutation matrix necessary to achieve this.

The current article presents results of the report [Sch02] obtainable via

`http://ist.unibw-muenchen.de/People/schmidt/DecompoHomePage.html`

which gives a detailed account of the topic. The report is not just a research report but also a Haskell program in literate style. In contrast, the present article only gives hints as to these programs. Therefore, some details are omitted.

This article is organized as follows. Chapter 2 presents the idea of extracting theories as proposed in this paper. Then Ch. 3 will mention some prerequisites. The hints concerning the relational language used are given in Ch. 4, followed by Ch. 5 with models and interpretations in Haskell. With Ch. 6 the first decomposition based on the strongly connected component ontology is elaborated in some detail to further clarify the idea. Theoretical basics of the more sophisticated Galois decompositions are explained in Ch. 7 before these are made ready for programming in Ch. 8.

[1] Cooperation and communication around this research was partly sponsored by the European COST Action 274: TARSKI (Theory and Application of Relational Structures as Knowledge Instruments), which is gratefully acknowledged.

2 The Idea of Theory Extraction

If some concrete relations are given as boolean matrices, one may talk about these in terms of a logical language and theory. We provide names for the relations and names for row and column entries which will be interpreted so that the name of the relation gets assigned the matrix as its interpretation, etc. This already gives us a sparse language and a sparse theory without any specific theorems to hold.

When a relational decomposition is reached, the language will have to contain also the necessary predicates and theorems expressing the algebraic idea behind the decomposition. Which predicates and theorems we need will depend on the decomposition we are aiming at and which we will call the given ontology. One ontology we have in mind is the game ontology. It considers the given homogeneous relation as the graph of a two-player game as described in [SS89,SS93], e.g. One wishes to solve the game, i.e., to qualify the positions as to being a position of win, of draw, or of loss for the player about to move. Other ontologies that have been handled include irreducibility, difunctionality, matching, etc.

The following diagram shows the idea. We start with the given model of the sparse theory (which is the relation originally given) and some ontology-enhanced theory. What we are constructing is in a sense the pushout.

$$\begin{array}{ccc} \text{Sparse theory} & \models & \text{Given model} \\ \downarrow & & \downarrow \\ \text{Ontology-enhanced theory} & \models & \text{Result model} \end{array}$$

After decomposition, the given model may be viewed with the mechanism our ontology has provided. Then the theory will also contain certain formulae describing what holds between the new ontology-dependent items.

Consider the following trivial example. We start with the left relation B and aim at the game ontology decomposition. Then we will obtain two uniquely determined sets a, b satisfying the relational theorems $a = \overline{B \cdot b}$ and $\overline{B \cdot a} = b$. In the game ontology they may be interpreted as *loss* $:= a$, *win* $:= \overline{b}$, and *draw* $:= b \cap \overline{a}$, below arranged as partition into *win, draw, loss*. They will furthermore allow us to obtain a permutation so as to rearrange the matrix B to the right-hand side form, which makes the algebraic laws easily visible.

$$\begin{array}{c} \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \end{array} \begin{array}{c} \begin{array}{ccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 \end{array} \\ \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 \end{pmatrix} \end{array} \quad \begin{array}{c} \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \end{array} \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} \end{array} \quad \begin{array}{c} \begin{array}{ccccccc} 2 & 4 & 3 & 6 & 7 & 1 & 5 \end{array} \\ \begin{array}{c} 2 \\ 4 \\ 3 \\ 6 \\ 7 \\ 1 \\ 5 \end{array} \left(\begin{array}{cc|ccc|cc} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ \hline 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ \hline 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 \end{array}\right) \end{array} \end{array}$$

Original game matrix, partition and rearranged matrix

3 Preliminaries

A major aspect in this work is, thus, to be able to handle permutations, to derive them for special purposes, to treat them as relations when appropriate, and to make them fully available in the Haskell program. Permutations may be considered as a function, decomposed into cycles, or as a permutation matrix. Everybody who has worked in functional programming will admit that transition from one form to the other is a simple programming task. As everything is worked out in the report [Sch02], we need not explain it here in detail. The additional general reference is [SS89,SS93], which will not be mentioned every time.

The permutations shall mostly be determined from partitions of a set so as to convert it to a list of elements with elements of an equivalence class side aside. Again, one will concede that this is a solvable task for a functional programmer; so we avoid mentioning this in detail here. One should, however, observe that sometimes rows and columns are permuted simultaneously and sometimes independently. These two situations need rather different treatment.

As space is limited, we cannot collect all the well-known and often recalled material in order to make this article self-contained. We restrict to mentioning that relations here are conceived as subsets $R \subseteq X \times Y$ of the Cartesian product of some sets X, Y between which they are defined to hold. Operations are composition $;$, transposition $^{\intercal}$, identities $\mathbb{I}$, union $\cup$, intersection $\cap$, complementation $\overline{\ }$, null relations $\mathbb{L}$, universal relations $\mathbb{T}$, and containment $\subseteq$ of relations.

When expressing the basic relational operators in Haskell, the programming language chosen, we will handle relations as rectangular boolean matrices. Often we represent their entries `True` by **`1`** and `False` by **`0`** when showing matrices in the text. The following basic relational operators `|||, &&&, ***, <==` for union, intersection, composition and containment of relations are all formulated in Haskell.

4 Relational Language

To formulate the theorems of an ontology-enhanced theory, we will need the language generated by the denotations of these operations, and we will not always immediately execute them. So we distinguish, e.g., between the denotation `:***:` as a Haskell infix constructor, and its interpretation, the operation `***`. Therefore, a relational language is presented allowing us to talk about elements, vectors (or sets), and (binary) relations. We are going to work in a typed or heterogeneous setting, which means that we start from a category.

```
data CatObject = Obj String | ...
```

So we will be able to give names to the category objects using the constructor `Obj` and a chosen string. The category will later stay the same when an extended theory is extracted guided by some ontology.

Then we need denotations for individual variables, constants, functions, and predicates. In our setting, we always bind these together with their typing, and we restrict to unary predicate constants which we call vectors and binary predicate constants, which we call relations. A relational constant is nothing more than a name, the string. Its type are the `CatObject`s between which the relation is supposed to hold. They are, however, not concretely given as we stay — so far — on the syntactical side.

```
data ElemConst = Elem String CatObject
data VectConst = Vect String CatObject
data RelaConst = Rela String CatObject CatObject
```

On all this, we now build first-order predicate logic, introducing individual variables, terms, and formulae.

```
data ElemVari = VarE String CatObject
data VectVari = VarV String CatObject
data RelaVari = VarR String CatObject CatObject
```

Vectors are here supposed to be column vectors. From the beginning, we distinguish element terms, vector terms, and relation terms. Null, universal, and identity relation constants may uniformly be denoted throughout as indicated.

```
data ElemTerm = EC ElemConst | EV ElemVari | ...
data VectTerm = VC VectConst | VV VectVari | RelaTerm :****: VectTerm |
                VectTerm :||||: VectTerm | VectTerm :&&&&: VectTerm |
                NegVect VectTerm | NullV CatObject | UnivV CatObject |
                VFctAppl RelaFct RelaTerm | ...
data RelaTerm = RC RelaConst | RV RelaVari | RelaTerm :***: RelaTerm |
                NegaRela RelaTerm | Transp RelaTerm | Ident CatObject |
                NullR CatObject CatObject | UnivR CatObject CatObject |
                RelaTerm :|||: RelaTerm | RelaTerm :&&&: RelaTerm |  ...
data ElemFct  = EFCT ElemVari ElemTerm
data VectFct  = VFCT VectVari VectTerm
data RelaFct  = RFCT RelaVari RelaTerm
```

The 0-ary operations for $\perp\!\!\!\perp, \amalg, \prod$ require giving domains or codomains, respectively. Typical checks such as `relaTermIsWellFormed, typeOfVectTerm` are provided for well-formedness, type control, etc.

In order to further facilitate maintenance of ubiquitous typing three forms of formulae are distinguished. In case $x \in v$, this is represented by the vector formula `VE v x`. In a similar way, `REE r x y` means $(x, y) \in r$.

```
data ElemForm = Equation ElemTerm  ElemTerm | ...
data VectForm = VectTerm :<===: VectTerm | VE VectTerm ElemTerm | ...
data RelaForm = RelaTerm :<==: RelaTerm |
                REE RelaTerm ElemTerm ElemTerm | ...
data Formula  = EF ElemForm | VF VectForm | RF RelaForm |
                Negated  Formula | Implies  Formula Formula |
                Disjunct Formula Formula | Conjunct Formula Formula | ...
```

Again, typing and well-formedness are defined as usual. The types of all kinds of formulae are intended to be `Bool`. Free variables are defined as usual, and then a theory (fragment) may be formulated as a data structure in Haskell.

```
data Theory = TH String          -- name of the theory
                 [CatObject]     -- carrier set denotations
                 [ElemConst]     -- element denotations
                 [VectConst]     -- subset denotations
                 [RelaConst]     -- relation denotations
                 [VectFct]       -- vector functions
                 [RelaFct]       -- relation functions
                 [Formula]       -- formulae demanded to hold
```

Of course, some testing is provided with `checkTheoryWellDefined`. The following is an example with just a denotation for a base set and a denotation for a relation intended to be defined on it, but only empty denotation lists `[]` provided for elements, subsets, functions, and formulae.

```
verySparseTheory = TH "Example" [o] [] [] [Rela "B" o o] [] [] []
                                                  where o = Obj "BaseSet"
```

5 Models and Interpretations

While we have so far only been concerned with syntax, we will now offer the opportunity to interpret the language, and the theories we have defined, in a model. Therefore, we show how an interpretation may be given. In our approach, theory and model are both represented in Haskell, so the distinction between the two will sometimes be difficult.

Via an interpretation, the objects get assigned sets in this model, however, we just mention the cardinalities of the sets. So — in a rather trivial sense — numbers can be viewed as names of the rows and columns. Also, vector and relation denotations are assigned concrete versions by the model.

```
data InterpreteObjs = Carrier  CatObject   Int
data InterpreteCons = InterCon ElemConst   Int
data InterpreteVect = InterVec VectConst  [Bool]
data InterpreteRela = InterRel RelaConst [[Bool]]
data InterpreteVFct = InterVFc VectFct   ([Bool]  ->  [Bool])
data InterpreteRFct = InterRFc RelaFct  ([[Bool]] -> [[Bool]])
```

Only in rare cases as, e.g., when studying rooted graphs with the root distinguished, will we have element constants. We provide an automatic interpretation for null relations, universal relations, and identity relations. Putting this together, a model is defined as follows:

```
data Model = MO String             -- name of the model
               [InterpreteObjs]    -- cardinalities of carrier sets
```

```
                [InterpreteCons]   -- numbers of corresponding elements
                [InterpreteVect]   -- subset-interpreting boolean vectors
                [InterpreteRela]   -- relation-interpreting matrices
                [InterpreteVFct]   -- interpreted vector functions
                [InterpreteRFct]   -- interpreted relation functions
```

We provide some generic mechanisms on the model side in order to check whether the sets in question are assigned to objects consistently by the interpretations. Lots of technicalities are necessary to ensure that this works as it is supposed to.

Before an interpretation is possible, we need valuations of the individual variables, i.e., an environment as a list of variable/value pairs.

```
type InterpreteElemVari = (ElemVari,   Int)
type InterpreteVectVari = (VectVari,  [Bool])
type InterpreteRelaVari = (RelaVari, [[Bool]])
type ElemValuations     = [InterpreteElemVari]
type VectValuations     = [InterpreteVectVari]
type RelaValuations     = [InterpreteRelaVari]
type Environment        = (ElemValuations,VectValuations,RelaValuations)
```

Using a rather primitive lookup function, we may then write

```
valuation env v = ...
```

to get the value of `v` in the environment `env`. Now terms and formulae may be interpreted according to the following examples.

```
interpreteVectTerm :: Model -> Environment -> VectTerm -> [Bool]
interpreteVectTerm m env vt =
   let MO _ os _ vs _ _ _ = m
       (evs,vvs,rvs) = env
   in  case vt of
          VFctAppl vf vt2 -> let ivf = interpreteVectFct  m env vf
                                 ivt = interpreteVectTerm m env vt2
                             in  ivf ivt
          ...

interpreteVectFct ::  Model -> Environment -> VectFct -> [Bool] -> [Bool]
interpreteVectFct m env vf bv =
   let VFCT vv vt = vf
       (evs,vvs,rvs) = env
       viWITHbm = (evs,(vv,bv) : vvs,rvs)
   in  interpreteVectTerm m viWITHbm vt

interpreteRelaTerm :: Model -> Environment -> RelaTerm -> [[Bool]]
interpreteRelaTerm m env rt =
   let MO _ _ _ _ rs = m
   in  case rt of
```

```
          RC rc           -> (\(InterRel _ b) -> b) $ head $
                             dropWhile (\(InterRel e _) -> rc /= e) rs
          rt1 :***: rt2   -> let int1 = interpreteRelaTerm m env rt1
                                 int2 = interpreteRelaTerm m env rt2
                             in  int1 *** int2
          ...

interpreteRelaForm :: Model -> Environment -> RelaForm -> Bool
interpreteRelaForm m env rf =
   case rf of
      rt1 :<==: rt2 -> let int1 = interpreteRelaTerm m env rt1
                           int2 = interpreteRelaTerm m env rt2
                       in  int1 <== int2
      ...
```

Once a model for a theory is given — which is at the same time finite as well as sufficiently small —, it will be possible to check the model property against the theory with `checkIsModelForTheory mo th`.

6 Strongly Connected Component Ontology

These concepts of language, theory, model, and theory extraction shall now be exemplified in a field for which the theoretical background is well-known. Starting from a homogeneous relation, we plan to permute rows and columns simultaneously.

Let some relation R be given and look for its reflexive transitive closure R^* and for the equivalence $R^* \cap R^{*\mathsf{T}}$ generated by this closure, the equivalence classes of which give the strongly connected components. Permuting rows and columns of R simultaneously so as to have them grouped according to these equivalence classes gives much insight into the structure of R.

First, a sparse theory is formulated, and the schema for an ontology-enhanced theory. In the sparse theory upon start, we simply know that a node set is given together with a relation R on it and no formulae are supposed to hold. So we provide denotations for a single category object `sO` and a relation constant `r` on it.

```
strongConnCompSparseTheory :: CatObject -> RelaConst -> Theory
strongConnCompSparseTheory sO r =
   TH "StrongConnCompSparseTheory" [sO] [] [] [r] [] [] []
```

Later, we will start with a model depending on a boolean matrix `gR`, the given relation, as a parameter.

```
strongConnCompGivenModel :: CatObject -> RelaConst -> [[Bool]] -> Model
strongConnCompGivenModel sO r gR =
   MO "StrongConnCompGivenModel" [Carrier sO (rows gR)]
                                 [] [] [InterRel r gR] [] []
```

No provisions have been made to denote single elements of the node set. As the number of connected components is not known beforehand, we provide the name for the list of partitioning connected components as non-empty subsets of entries as `pl`. The list of vector denotations of the ontology is then used in the formulae that describe what has been achieved when decomposition is executed once the model is known.

```
strongConnCompOntolEnhancedTheorySchema ::
                   CatObject -> RelaConst -> [VectConst] -> Theory
strongConnCompOntolEnhancedTheorySchema sO r pl =
   let rt  = RC r
       d   = domRC r
       vts = map VC pl
       nullVect = NullV d
       subsetNonEmpty s = Negated (VF (s :<===: nullVect))
       partitionSetsNonEmpty = map subsetNonEmpty vts
       subsDisjoint (s1,s2) = VF (s1 :&&&&: s2 :<===: nullVect)
       allUnordPairsOfPartSets s =
          let fff res [] = res
              fff res (h:t) = fff (res ++ (map (\x -> (h,x)) t)) t
          in  fff [] s
       partSetsDisjoint = map subsDisjoint (allUnordPairsOfPartSets vts)
       syntSetUnion = foldr (:||||:) nullVect vts
       partitionSetsExhaust = VF (syntSetUnion :====: (UnivV d))
       rEquivalenceClosure = Rela "rEquCl" sO sO
       equivalenceTimesSetEqualsSet s =
          VF (RC rEquivalenceClosure :****: s :====: s)
   in  TH "StrongConnCompOntolEnhancedTheory" [sO] [] pl
          [r,rEquivalenceClosure] [] []
          (partitionSetsNonEmpty ++ partSetsDisjoint ++
          [partitionSetsExhaust] ++ map equivalenceTimesSetEqualsSet vts)
```

The piece of code schematically generates the formulae satisfied by strongly connected components, namely that they be nonempty, disjoint, and exhaust the set. In addition, they are closed with respect to the equivalence formed of the reflexive-transitive closure intersected with its transpose. We have not included the properties the equivalence closure enjoys, as these are standard.

This schema of a theory cannot be checked for well-formedness with `checkTheory-WellDefined`, as it is composed of lists of sets and formulae with lengths not yet determined. After applying the decomposition algorithm, there will be the result model where subset constants are filled in corresponding to the sets of the partition as long as there exist further strongly connected components.

```
strongConnCompResultModel ::
           Theory -> Model -> ([VectConst] -> Theory) -> (Theory,Model)
strongConnCompResultModel sparseTheory givenModel enhancedTheorySchema =
   let TH _ [co] _ _ [rc] _ _ _ = sparseTheory
       MO _ iC _ _ iR _ _ = givenModel
```

```
      eThS = enhancedTheorySchema
      r = interpreteRelaConst givenModel ([],[],[]) rc
      reflTransClos = reflTranClosure r
      eq = transpMat reflTransClos &&& reflTransClos
      rowTypesH     = sort $ nub reflTransClos
      rowTypesH1 [] = []
      rowTypesH1 (hh:tt) = hh : (rowTypesH1
                    (map (\ pp -> zipWith (&&) (map not hh) pp) tt))
      rowTypes  = reverse $ rowTypesH1 rowTypesH
      partitionSetsNamed =
         zipWith (\a b -> Vect ("partSet" ++ show a) co) [1..] rowTypes
      th@(TH _ _ _ _ [_,rEqCl] _ _ _) = eThS partitionSetsNamed
      partitionSetsInterpreted =
         zipWith (\a b -> InterVec a b) partitionSetsNamed rowTypes
      resModel = MO "StrongConnCompResultModel" iC []
         partitionSetsInterpreted (iR ++ [InterRel rEqCl eq]) [] []
in    (th,resModel)
```

Here, the three parameters of the pushout are taken and the reflexive-transitive closure is formed, as well as the equivalence defined by it. Then the rows of the closure are considered. First, duplicates are eliminated, then the rows are sorted in order to have the matrix later with small elements first followed by greater ones. As now the number of strongly connected components is known, they may be named as $\texttt{partSet}_i$, followed by building the enhanced theory. To these names are then attached the resulting partition sets and delivered in the result model. Once `gR` is concretely given as the matrix on the left, a sophisticated TEX-generating matrix printing algorithm will produce the subdivided matrix on the right showing the connected components.

	1	2	3	4	5	6	7	8	9	10	11	12	13
1	1	0	0	0	0	0	0	0	0	0	0	0	1
2	0	0	0	0	0	0	0	1	0	0	0	1	1
3	0	0	1	0	0	1	0	0	0	1	0	0	0
4	0	0	0	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0	1	0	0
6	0	0	0	0	0	1	0	0	0	0	0	1	0
7	0	1	0	0	0	0	0	0	1	0	0	0	0
8	0	1	0	0	0	0	1	0	0	0	0	0	0
9	0	0	0	0	1	0	0	0	1	0	0	0	0
10	0	0	0	0	0	0	0	0	1	0	0	0	0
11	0	0	0	0	0	0	0	0	1	1	0	0	0
12	1	0	1	0	0	0	0	0	0	0	0	0	0
13	0	0	1	0	0	0	0	0	0	0	1	0	0

	2	7	8	1	3	6	12	13	4	5	9	10	11
2	0	0	1	0	0	0	1	1	0	0	0	0	0
7	1	0	0	0	0	0	0	0	0	0	1	0	0
8	1	1	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	1	0	0	0	1	0	0	0	0	0
3	0	0	0	0	1	1	0	0	0	0	0	1	0
6	0	0	0	0	0	1	1	0	0	0	0	0	0
12	0	0	0	1	1	0	0	0	0	0	0	0	0
13	0	0	0	0	1	0	0	0	0	0	0	0	1
4	0	0	0	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0	0	0	1
9	0	0	0	0	0	0	0	0	0	1	1	0	0
10	0	0	0	0	0	0	0	0	0	0	1	0	0
11	0	0	0	0	0	0	0	0	0	0	1	1	0

An original relation and the rearranged relation

One will observe that all this is based on the following

Proposition. Any given finite homogeneous relation R can by simultaneously permuting rows and columns be transformed into a matrix of the following form: It has upper triangular pattern with square diagonal blocks

$$\begin{pmatrix} \square & * & * & * \\ \bot\!\!\bot & \square & * & * \\ \bot\!\!\bot & \bot\!\!\bot & \square & * \\ \bot\!\!\bot & \bot\!\!\bot & \bot\!\!\bot & \square \end{pmatrix}$$

where $* = \bot\!\!\bot$ unless the generated preorder R^* allows entries $\neq \bot\!\!\bot$. The reflexive-transitive closure of every diagonal block is the universal relation $\top\!\!\top$. $\square$

7 Galois Decompositions

We now present some more involved possibilities to decompose a relation. They are closely related, as all of them may be formulated using a Galois correspondence. Afterwards a schema for the decompositions will be given that enables us to handle them more or less simultaneously.

In all of these cases, we will need two antitone mappings between powersets, which we call $\sigma : \mathcal{P}(V) \to \mathcal{P}(W)$ and $\pi : \mathcal{P}(W) \to \mathcal{P}(V)$. These mappings are usually determined by a relational construct based on some relation $B : V \leftrightarrow W$. Nested iterations will then start with the empty subset of V on the left and the full subset of W on the right — or vice versa. While there is a lot of theory necessary for the infinite case, the finite case is rather simple. Consider the starting configuration with its trivial containments $\bot\!\!\bot \subseteq \pi(\top\!\!\top)$ and $\sigma(\bot\!\!\bot) \subseteq \top\!\!\top$ which are perpetuated by the antitone mappings to $\bot\!\!\bot \subseteq \pi(\top\!\!\top) \subseteq \pi(\sigma(\bot\!\!\bot)) \subseteq \ldots$ and $\ldots \subseteq \sigma(\pi(\top\!\!\top)) \subseteq \sigma(\bot\!\!\bot) \subseteq \top\!\!\top$. In the finite case, these two sequences will eventually become stationary. The effect of the iteration is that the least fixed point a of $v \mapsto \pi(\sigma(v))$ on the side started with the empty set is related to the greatest fixed point b of $w \mapsto \sigma(\pi(w))$ on the side started from the full set. The final situation obtained will be characterized by $a = \pi(b)$ and $\sigma(a) = b$.

7.1 Termination

The set of all points of a graph, from which only paths of finite length emerge,

$$J(R) := \mathtt{inf}\,\{\, x \mid \overline{x} = R \mathbin{;} \overline{x} \,\}$$

is called the **initial part** $J(R)$ of the relation R underlying the graph. We are going to determine the initial part of that relation. A relation is *progressively finite* if $\overline{J(R)} = \top\!\!\top$. A slightly different property is being *progressively bounded*, $\mathtt{sup}_{h \geq 0} \overline{B^h \mathbin{;} \top\!\!\top} = \top\!\!\top$. A difference between the two exists only for non-finite relations; it may, thus, be neglected here.

Looking at the definition of the initial part, the two antitone functionals $v \mapsto \sigma(v) := \overline{v}$ and $w \mapsto \pi(w) := R \mathbin{;} \overline{w}$ seem to play a major role. One will later easily identify them in `Constituents`.

The algorithm applied to the relation R will result in a pair (a, b) of vectors. The relational formulae valid for the final pair (a, b) of the iteration are $a = \pi(b) = R;\overline{b}$ and $b = \sigma(a) = \overline{a}$. (In this case, it is uninteresting to start with the empty set and the full set exchanged from left to right.)

Here, b is the initial part belonging to R: There are no paths of infinite length from the vertices of b, which, however, do exist starting from vertices of a. This is based on the following

Proposition. Any finite homogeneous relation may by simultaneously permuting rows and columns be transformed into a matrix satisfying the following basic structure with square diagonal entries:

$$\begin{pmatrix} \text{progressively bounded} & \text{⫫} \\ * & \text{total} \end{pmatrix} \qquad \square$$

This subdivision into groups "initial part/infinite path exists" is uniquely determined, and indeed

$$a = \begin{pmatrix} \text{⫫} \\ \text{⫪} \end{pmatrix} = \begin{pmatrix} \text{progressively bounded} & \text{⫫} \\ * & \text{total} \end{pmatrix} ; \overline{\begin{pmatrix} \text{⫪} \\ \text{⫫} \end{pmatrix}}, \quad b = \begin{pmatrix} \text{⫪} \\ \text{⫫} \end{pmatrix} = \overline{\begin{pmatrix} \text{⫫} \\ \text{⫪} \end{pmatrix}}$$

The termination-oriented decomposition may prove useful in the following case: Assume a preference relation being given, where it is not clear from the beginning that this preference is circuit-free. There is a tendency of ranking equal all those who belong to a circuit. The initial part collects all items from which one will not run into a circuit at all, so that they are properly ranked by the given relation. The others should be treated with the strongly connected component ontology and then be ranked groupwise.

	1	2	3	4	5	6	7	8	9	10	11
1	0	1	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	1	0
3	0	1	1	1	0	1	0	0	0	0	0
4	0	0	0	1	1	0	0	0	0	0	0
5	0	1	0	0	0	1	1	0	0	1	0
6	1	0	1	0	0	0	0	0	0	0	0
7	0	0	1	1	0	0	0	0	0	0	0
8	0	0	0	0	1	0	0	0	0	0	0
9	0	0	0	0	0	1	0	1	0	0	0
10	0	0	0	0	0	0	0	0	0	0	0
11	1	1	0	1	0	0	0	0	0	0	0

	1	2	10	3	4	5	6	7	8	9	11
1	0	1	0	0	0	0	0	0	0	0	0
2	0	0	1	0	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0	0	0	0
3	0	1	0	1	1	0	1	0	0	0	0
4	0	0	0	0	1	1	0	0	0	0	0
5	0	1	1	0	0	0	1	1	0	0	0
6	1	0	0	1	0	0	0	0	0	0	0
7	0	0	0	1	1	0	0	0	0	0	0
8	0	0	0	0	0	1	0	0	0	0	0
9	0	0	0	0	0	0	1	0	1	0	0
11	1	1	0	0	1	0	0	0	0	0	0

A relation, original and rearranged according to its initial part

7.2 Matching and Assignment

A second Galois decomposition is known to exist in connection with matchings and assignments. Here we will for the first time consider heterogeneous relations.

Let two matrices $Q, \lambda : V \leftrightarrow W$ be given, where $\lambda \subseteq Q$ is univalent and injective, i.e. a matching — possibly not yet of maximum cardinality, for instance

$$Q = \begin{array}{c} \begin{matrix} & 1 & 2 & 3 & 4 & 5 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \end{matrix} \begin{pmatrix} \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} \\ \mathbf{0} & \mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{1} \\ \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{1} \end{pmatrix} \end{array} \quad \supseteq \quad \lambda = \begin{array}{c} \begin{matrix} & 1 & 2 & 3 & 4 & 5 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \end{matrix} \begin{pmatrix} \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} \end{pmatrix} \end{array}$$

Sympathy and matching

We consider Q to be a relation of sympathy between a set of boys and a set of girls and λ the set of current dating assignments, assumed only to be established if sympathy holds. We now try to maximize the number of dating assignments.

Definition. i) Given a possibly heterogeneous relation Q, the relation λ will be called a **Q-matching** if it is univalent, injective, and contained in Q, i.e., if

$$\lambda \subseteq Q \qquad \lambda ; \lambda^{\mathsf{T}} \subseteq \mathbb{I}, \qquad \lambda^{\mathsf{T}} ; \lambda \subseteq \mathbb{I}.$$

ii) We say that a point set x can be **saturated** if there exists a matching λ with $\lambda ; \mathbb{T} = x$. □

The current matching λ may have its origin from a procedure like the following that assigns matchings as long as no backtracking is necessary. The second parameter of the encapsulated function serves for accounting purposes so that no matching row will afterwards contain more than one assignment.

```
trivialMatchAbove q lambda =
   let colsOccupied = map or (transpMat lambda)
       trivialMatchRow []       []         = []
       trivialMatchRow (True:t) (False:_ ) =
          True :(replicate (length t) False)
       trivialMatchRow (_   :t) (_    :tf) = False:(trivialMatchRow t tf)
       trivialMatchAboveH []             _ = []
       trivialMatchAboveH ((hq, hl) : t) f =
             let actRow = case or hl of
                             True  -> hl
                             False -> trivialMatchRow hq f
                 fNEW   = zipWith (||) actRow f
             in  actRow : (trivialMatchAboveH t fNEW)
   in  trivialMatchAboveH (zip q lambda) colsOccupied
```

Given this setting, it is again wise to design two antitone mappings. The first shall relate a set of boys to those girls not sympathetic to anyone of them,

$v \mapsto \sigma(v) = \overline{Q^\intercal ; v}$. The second shall present the set of boys not assigned to some set of girls, $w \mapsto \pi(w) = \overline{\lambda ; w}$.

The iteration will end with two vectors (a, b) satisfying $a = \pi(b)$ and $\sigma(a) = b$ as before. Here, this means $\overline{a} = \lambda ; b$ and $\overline{b} = Q^\intercal ; a$. In addition $\overline{a} = Q ; b$. This follows from the chain $\overline{a} = \lambda ; b \subseteq Q ; b \subseteq \overline{a}$, which implies equality at every intermediate state. Only the resulting equalities for a, b have been used together with monotony and the Schröder rule.

One may discuss whether we had been right in deciding for starting the iteration procedure with $\mathbb{L}$ on the left side and $\mathbb{T}$ on the right. Assume we had decided the other way round. This would obviously mean the same as starting as before, but with Q, λ transposed. Instead of $\overline{a} = \lambda ; b$, $\overline{b} = Q^\intercal ; a$, and $\overline{a} = Q ; b$ we would then obtain the three conditions with Q replaced by $Q^\intercal$, λ by $\lambda^\intercal$, and a, b exchanged. While the two equations with Q just exchange each other, the first is transferred to $\overline{b} = \lambda^\intercal ; a$. This means that the resulting decomposition of the matrices does *not* depend on the choice — if this fourth equation is also satisfied.

It is thus not uninteresting to concentrate on condition $\overline{b} = \lambda^\intercal ; a$. After having applied `trivialMatch` to some sympathy relation and applying the iteration, it may not yet be satisfied. So let us assume $\overline{b} = \lambda^\intercal ; a$ *not* to hold, which means that $\overline{b} = Q^\intercal ; a \supsetneqq \lambda^\intercal ; a$.

We make use of the formula $\lambda ; \overline{S} = \lambda ; \mathbb{T} \cap \overline{\lambda ; S}$, which holds since λ is univalent. The iteration ends with $\overline{b} = Q^\intercal ; a$ and $\overline{a} = \lambda ; b$. This easily expands to

$$\overline{b} = Q^\intercal ; a = Q^\intercal ; \overline{\lambda ; b} = Q^\intercal ; \overline{\lambda ; \overline{Q^\intercal ; a}} = Q^\intercal ; \lambda ; \overline{Q^\intercal ; \overline{\lambda ; \overline{Q^\intercal ; a}}} \ \ldots$$

from which the last but one becomes

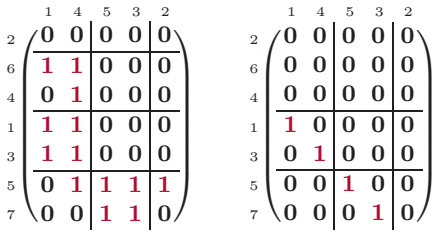
$$\begin{aligned} \overline{b} = Q^\intercal ; a = Q^\intercal ; \overline{\lambda ; b} &= Q^\intercal ; \overline{\lambda ; \mathbb{T} \cap \overline{\lambda ; Q^\intercal ; a}} = Q^\intercal ; (\overline{\lambda ; \mathbb{T}} \cup \lambda ; Q^\intercal ; a) \\ &= Q^\intercal ; (\overline{\lambda ; \mathbb{T}} \cup \lambda ; Q^\intercal ; (\overline{\lambda ; \mathbb{T}} \cup \lambda ; Q^\intercal ; a)) \end{aligned}$$

indicating how to prove that

$$\overline{b} = (Q^\intercal \cup Q^\intercal ; \lambda ; Q^\intercal \cup Q^\intercal ; \lambda ; Q^\intercal ; \lambda ; Q^\intercal \cup \ \ldots) ; \overline{\lambda ; \mathbb{T}}$$

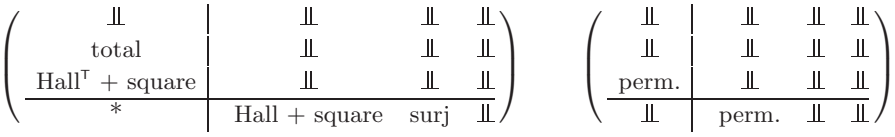
If $\lambda^\intercal ; a \subsetneqq \overline{b}$, we may thus find a point in $\overline{\lambda^\intercal ; a} \cap (Q^\intercal \cup Q^\intercal ; \lambda ; Q^\intercal \cup Q^\intercal ; \lambda ; Q^\intercal ; \lambda ; Q^\intercal \cup \ \ldots) ; \overline{\lambda ; \mathbb{T}}$ which leads to the famous alternating chain algorithm. While `trivialMatch` didn't do any backtracking, the alternating chain algorithm does. It therefore delivers cardinality maximum matchings and not just matchings that cannot be increased by finding an enclosing one.

We now visualize the results of this matching iteration by concentrating on the subdivision of the matrices Q, λ initially considered by the resulting vectors $a = \{2, 6, 4, 1, 3\}$ and $b = \{5, 3, 2\}$. One easily proves that $\overline{b} = \lambda^\intercal ; a$ is already satisfied. Some additional care must be taken concerning empty rows or columns in Q. To obtain the subdivided relations neatly, these are placed at the beginning of the rows, respectively at the end of the columns. In addition, rows and columns may be permuted so as to let λ appear as a diagonal.



Sympathy and matching rearranged

Proposition. Any given heterogeneous relation Q admits a cardinality maximum matching $\lambda \subseteq Q$. Both relations may then in addition be simultaneously transformed into matrices of the following form by independently permuting rows and columns: Principally, they have a 4 by 4 pattern with possibly empty zones and not necessarily square diagonal blocks.



- The first zone of rows and the last zone of columns of both matrices are zero rows resp. columns.
- The upper right 3 by 3 zones are again empty.
- The zones $\lambda_{3,1}$ and $\lambda_{4,2}$ are identity matrices.
- Zone $Q_{2,1}$ is a total relation.
- Zone $Q_{4,3}$ is a surjective relation.

The cardinality maximum matching λ is not uniquely determined, only by cardinality. The decomposition is, thus, uniquely determined up to the so-called term-rank, defined below, which here shows up as the total length of the diagonal in λ. □

We also provide the definition of term-rank and Hall-condition.

Definition. i) Given a relation Q, the **term rank** is defined as the minimum number of lines (i.e., rows or columns) necessary to cover all entries **1** in Q, i.e.

$$\mathtt{min}\{|s| + |t| \mid Q \mathbin{;} \overline{t} \subseteq s\}.$$

ii) Given a relation Q and a set x, we say that x satisfies the **Hall condition**

$$\iff \quad |z| \leq |Q^\intercal \mathbin{;} z| \text{ for every subset } z \subseteq x. \qquad \square$$

8 Galois Decomposition Ontologies

Appropriate ontologies for these Galois decompositions shall now be developed. They may serve to solve a diversity of application problems, such as matching, line-covering, assignment, games, etc.

This shall be done simultaneously, i.e., in a schema that may be instantiated later to cope with these variants. So we give the task in a schematic form also and introduce the following constituents as a parameter.

```
type Constituents = (String,[CatObject],[RelaConst],[VectFct])
gameConstituents :: Constituents
gameConstituents =
   let singleObject = OC (CstO "NodeSet")
       b  = Rela "B" singleObject singleObject
       vv = VarV "v" singleObject
       f  = VFCT vv (NegVect (RC b :****: (VV vv)))
   in  ("Game",[singleObject],[b],[f,f])
terminationConstituents :: Constituents
terminationConstituents =
   let singleObject = OC (CstO "NodeSet")
       b = Rela "B" singleObject singleObject
       vv1 = VarV "v1" singleObject
       vv2 = VarV "v2" singleObject
       f1 = VFCT vv1 (NegVect (VV vv1))
       f2 = VFCT vv2 (RC b :****: (NegVect (VV vv2)))
   in  ("Termination",[singleObject],[b],[f1,f2])
matchAssignConstituents :: Constituents
matchAssignConstituents =
   let firstObject = OC (CstO "NodeSet1")
       secndObject = OC (CstO "NodeSet2")
       q      = Rela "B"      firstObject secndObject
       lambda = Rela "Lambda" firstObject secndObject
       vv1 = VarV "v1" firstObject
       vv2 = VarV "v2" secndObject
       f1 = VFCT vv1 (NegVect (RC q      :****: (VV vv1)))
       f2 = VFCT vv2 (NegVect (RC lambda :****: (VV vv2)))
   in  ("MatchAssign",[firstObject,secndObject],[q,lambda],[f1,f2])
```

As can be seen, we have provided for denotations for category objects, relation constants, and antitone functions relating vectors on the domain side to vectors on the codomain side and vice versa. From this, we get the sparse theory in a schematic way. We may afterwards instantiate with the respective constituents.

```
sparseTheorySchema :: Constituents -> Theory
sparseTheorySchema cs =
   let (s,os,rs,vfs) = cs
   in  TH ("SparseTheory" ++ s) os [] [] rs [] [] []
sparseGameTheory        = sparseTheorySchema gameConstituents
```

```
sparseTerminationTheory = sparseTheorySchema terminationConstituents
sparseMatchAssignTheory = sparseTheorySchema matchAssignConstituents
```

Also the given models may be presented schematically, providing the respective constituents first and then the list of given matrices.

```
givenModelSchema :: Constituents -> [ [[Bool]] ] -> Model
givenModelSchema cs gRs =
   let TH s os _ _ rs _ _ _ = sparseTheorySchema cs
       irs = zipWith (\ rc bm -> InterRel rc bm) rs gRs
       osM = map (\(a,b) -> Carrier a b) $ nub $ concat $
             map (\(InterRel rc bm) -> [(domRC rc,rows bm),
                                        (codRC rc,cols bm)]) irs
   in  MO ("GivenModel" ++ s) osM [] [] irs [] []
givenModelGame       gRs = givenModelSchema gameConstituents        gRs
givenModeTermination gRs = givenModelSchema terminationConstituents gRs
givenModeMatchAssign gRs = givenModelSchema matchAssignConstituents gRs
```

With, e.g.,
`givenModelGame = givenModelSchema gameConstituents listOfMat`
this may be instantiated. We can immediately check
`checkIsModelForTheory givenModelGame sparseGameTheory`,
for instance.

As there will always be a result which simply subdivides the domain as well as the range set into two subsets, it is a feasible task to find the schema of a Galois ontology-enhanced theory.

```
galoisOntolEnhancedTheorySchema :: Constituents -> Theory
galoisOntolEnhancedTheorySchema cs =
   let (_,_,_,vfs) = cs
       TH s os es vs rs _ _ fs = sparseTheorySchema cs
       rsTerm  = map RC rs
       [lr,rl] = vfs
       (d,c)   = typeOfFV lr

       leftFixAboveNull  = Vect "LeftFixAboveNull"  d
       rightFixBelowUniv = Vect "RightFixBelowUniv" c

       leftFixAboveNullT  = VC leftFixAboveNull
       rightFixBelowUnivT = VC rightFixBelowUniv

       sigmaLeftNullEqualsRightUniv =
          VF $ VFctAppl lr leftFixAboveNullT  :====: rightFixBelowUnivT
       leftNullEqualsPiRightUniv    =
          VF $ VFctAppl rl rightFixBelowUnivT :====: leftFixAboveNullT

   in  TH ("GaloisEnhancedTheoryTo" ++ s) os es
          [leftFixAboveNull,rightFixBelowUniv]
          rs vfs [] (fs ++
             [leftNullEqualsPiRightUniv,sigmaLeftNullEqualsRightUniv])
```

There is not much computation in this piece of code. Two vector denotations are provided for and made to vector constants terms. Then the formula is built that says that appplying the left-right function to the left vector will result in the right vector. Finally the formula is generated that the right-left function applied to the right vector will result in the left vector. As before, instantiation is possible, e.g.,

```
gameOntologyEnhancedTheory =
   galoisOntolEnhancedTheorySchema gameConstituents
```

Then we develop the result model in a schematic form. The antitone functions have to be inserted as appropriate. We formulate the basic iteration for the antitone functions along the well-known `until`-construct of Haskell with `lr` for σ and `rl` for π.

```
untilGalois lr rl (v, w)
     = let lrv = lr v
           rlw = rl w
       in  if (w == lrv) && (v == rlw) then (v, w)
                                       else untilGalois lr rl (rlw, lrv)
```

This `untilGalois` is the main algorithmic part in generating the result model in a schematic form. All the rest is designed to administrative purposes of getting the left-right and right-left functions appropriately out of the enhanced theory, interpreting them, and applying them, e.g.

```
galoisResultModelSchema :: Constituents -> [ [[Bool]] ] -> (Theory,Model)
galoisResultModelSchema cs gRs =
   let th@(TH s os _ [leftFixAboveNull,rightFixBelowUniv]
            rs vfs _ [lNP,sLN]) = galoisOntolEnhancedTheorySchema cs
       mo@(MO _ osM _ _ irs _ _) = givenModelSchema cs gRs
       [lR,rL] = vfs
       (d,c) = typeOfFV lR
       argVectConstLeft  = VarV "argL" d
       argVectConstRight = VarV "argR" c
       dSize = getObjectCarrierSize osM d
       cSize = getObjectCarrierSize osM c
       lr v = interpreteVectTerm mo ([],[(argVectConstLeft ,v)],[])
                 (VFctAppl lR (VV argVectConstLeft ))
       rl w = interpreteVectTerm mo ([],[(argVectConstRight,w)],[])
                 (VFctAppl rL (VV argVectConstRight))
       (leNuPi,siLeNu) = untilGalois lr rl (replicate dSize False,
                                            replicate cSize True)
       vectInterpretations = [InterVec leftFixAboveNull  leNuPi,
                              InterVec rightFixBelowUniv siLeNu]
       resModel = MO ("GaloisResultPushoutOf" ++ s)
                     osM [] vectInterpretations irs [(InterVFc lR lr),
                                                     (InterVFc rL rl)] []
   in  (th,resModel)
```

Instantiation is possible to

```
resGameModel =
   galoisResultModelSchema gameConstituents listOfMat
resTermionationModel =
   galoisResultModelSchema terminationConstituents listOfMat
resMatchAssignModel =
   galoisResultModelSchema matchAssignConstituents listOfMat
```

and one may check

```
isResultModelGame =
   checkIsModelForTheory mod the where (the,mod) = resGameModel
```

9 Conclusion and Outlook

We have provided several ontologies in which to embed newly presented relations for handling them in a pre-formatted way. With the methods presented it is possible to analyze a given relation with regard to different concepts and to visualize the results. This paper is in some regard related to work such as [Kit93,DL01,BR96]. Ordering decompositions have been studied using a similar technique in [Win03].

We hope that this will lead to future research. We have scanned a diversity of topics for their algebraic properties. On several occasions, we have replaced counting arguments by algebraic ones. Our hope is that these algebraic properties will be of value in handling fuzzy relations in this way, which do not lend themselves readily to counting methods.

In the course of this research, a wide-spectrum relational reference language [Sch03] far beyond the hints given here has been and is still being developed. It is conceived as part of the research of Work Area 2 *Mechanization* of the European COST Action TARSKI (*Theory and Applications of Relational Structures as Knowledge Instruments*) which attempts jointly to find ways to mechanize relational reasoning. Colleagues are expressly invited to take part in this endeavor and to further contribute to the design of the language.

Acknowledgments
Discussions with Michael Ebert, Eric Offermann, and Michael Winter provided considerable help.

References

[BR96] R. B. Bapat and T. E. S. Raghavan. *Nonnegative Matrices and Applications*, volume 64 of *Encyclopaedia of Mathematics and its Applications*. Cambridge University Press, 1996.

[DL01] Sašo Džeroski and Nada Lavrač, editors. *Relational Data Mining*. Springer-Verlag, 2001.

[Kit93] Leonid Kitainik. *Fuzzy Decision Procedures With Binary Relations — Towards a Unified Theory*, volume 13 of *Theory and Decision Library, Series D: System Theory, Knowledge Engineering and Problem Solving*. Kluwer Academic Publishers, 1993.

[Sch02] Gunther Schmidt. Decomposing Relations — Data Analysis Techniques for Boolean Matrices. Technical Report 2002-09, Fakultät für Informatik, Universität der Bundeswehr München, 2002. http://ist.unibw-muenchen.de/People/schmidt/DecompoHomePage.html, 79 pages.

[Sch03] Gunther Schmidt. Relational Language. Technical Report 2003-05, Fakultät für Informatik, Universität der Bundeswehr München, 2003. 101 pages.

[SS89] Gunther Schmidt and Thomas Ströhlein. *Relationen und Graphen*. Mathematik für Informatiker. Springer-Verlag, 1989. ISBN 3-540-50304-8, ISBN 0-387-50304-8.

[SS93] Gunther Schmidt and Thomas Ströhlein. *Relations and Graphs — Discrete Mathematics for Computer Scientists*. EATCS Monographs on Theoretical Computer Science. Springer-Verlag, 1993. ISBN 3-540-56254-0, ISBN 0-387-56254-0.

[Win03] Michael Winter. Decomposing Relations Into Orderings. In *Participants Proc. of the International Workshop RelMiCS '7 Relational Methods in Computer Science and 2nd International Workshop on Applications of Kleene Algebra, in combination with a workshop of the COST Action 274: TARSKI*, pages 190–196, 2003.