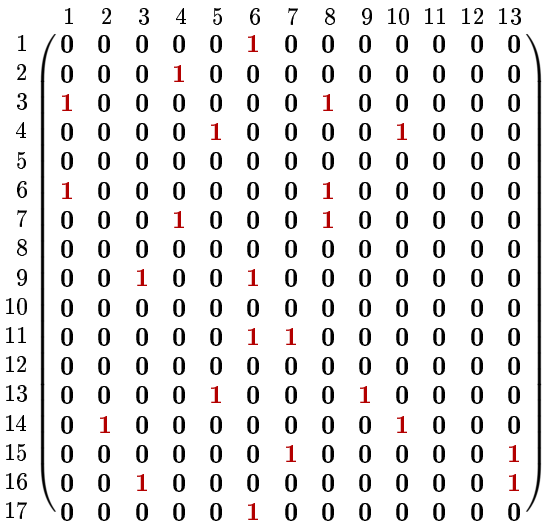
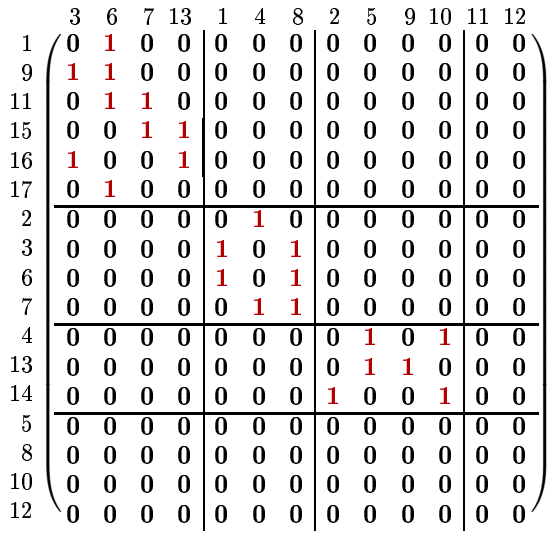
Decomposing Relations

Data Analysis Techniques for Boolean Matrices

GUNTHER SCHMIDT





BERICHT NR. 2002-09

DEZEMBER 2002

Universität der Bundeswehr München

Fakultät für

INFORMATIK

Werner-Heisenberg-Weg 39 • D-85577 Neubiberg

Decomposing Relations

Data Analysis Techniques for Boolean Matrices

Gunther Schmidt

Institute for Software Technology
Department of Computing Science
Federal Armed Forces University Munich

e-Mail: Schmidt@Informatik.UniBw-Muenchen.DE

February 23, 2003

Abstract

Known and new methods of decomposing a boolean relation are presented together with methods of making the decomposition visible. Homogeneous and heterogeneous relations are handled with non-iterative as well as iterative methods.

Such aspects as reducibility, cyclicity, primitivity, difunctionality, Ferrer's relations, Moore-Penrose inverses, independence and line-covering, chainability, game decompositions, matchings, Hall conditions, term rank, chainability, full indecomposability, and others are handled under one common roof.

We have also tried to collect several concepts for nonnegative real-valued matrices and to treat them as concepts for boolean matrices. An additional impetus for this study was to give all this a relation algebraic basis avoiding counting arguments. Several proofs of already known facts are, therefore, quite different from the classical ones.

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.

Contents

1 Introduction

This paper deals with several techniques to decompose matrices according to certain criteria. In small examples this can easily be done, while for bigger ones some of the techniques to be mentioned are asymptotically inefficient. We don't really care here for asymptotical efficiency as the applications we have in mind are limited in size. Rather, we consider this text as a support for teaching and for getting insight, as it gives visual help in many cases.

One area of applications is multicriteria decision making. There, relations are given beforehand and one asks for dichotomies generated by the relation [Kit93, DL01] following certain rational ideas. One may call this theorem extraction or theorem formulation — as opposed to theorem proving. Once formulated, theorem provers would certainly establish the theorem. But in practical situations it is more important *to find* the theorem on which one is willing to base decisions.

The key concept in these cases is the notion of a decision procedure on binary relations. So-called rationality concepts are defined. Then from given relations the "most rational" subsets on the domain side shall be evaluated. In decision analysis a diversity of choice rules is investigated, all of them leading to a partition of the set on which the preference relation holds.

Much of the basic approach can be demonstrated by the example of rearranging a matrix which is endowed with a specific property. Consider the following relation A. It is not impossible to convince oneself that it obeys $A ; A^\mathsf{T} ; A = A$, but one will not immediately grasp what this algebraic condition really means.

$$A = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \\ 10 \\ 11 \\ 12 \end{array} \begin{array}{c} \begin{array}{cccccccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 \end{array} \\ \begin{pmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 \end{pmatrix} \end{array}$$

$$\mathtt{rs} = \begin{pmatrix} 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{pmatrix}$$

$$\mathtt{cs} = \begin{pmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

A relation and the occurrences of its row and column types

In many cases results of some investigation automatically bring information that might be used for a partition of the set of rows and the set of columns, respectively. In this case, a partition into groups of identical rows and columns is easily obtained. It is a good idea to permute rows and columns so as to have the identical rows of the groups side aside. This means to independently permute rows and columns. The result then makes the essence of the algebraic condition $A ; A^\mathsf{T} ; A = A$ directly visible. Also the permutations are given in the result.

$$A_{\text{rearranged}} = \begin{array}{c|cccccccc|cccc|cc} & 1 & 3 & 5 & 8 & 9 & 11 & 12 & 14 & 2 & 4 & 7 & 10 & 6 & 13 \\ \hline 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 4 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 6 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 10 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 12 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ \hline 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 5 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 7 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ \hline 3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 9 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 11 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ \hline 8 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}$$

The relation rearranged according to its row and column types

After this initial example, one basic task to solve is as follows. Let a relation A be given, together with an indication `rs` which rows as well as an indication `cs` which columns are equal[1]. From this, a rearranged matrix resembling these partitions shall be computed as well as the permutation matrix necessary to achieve this.

The current report strives to provide the methodological basis together with a toolkit for the task of decomposing relations in various application disciplines. It is devoted to methods and techniques, starting from which it should be possible to build customised tools. So applications here play the role that the tools are expected to be applicable to a subset of them. We do not go into detail on applications; rather, we look for an in some way coherent presentation of techniques.

This report is organized as follows. Chapter 2 presents the prerequisites and tools used in this work, namely Haskell as a programming language, some techniques to handle permutations, and to derive them from partitions. We cannot avoid to recall some theory on relations in Chapter 3. Only the section on congruences contains novel approaches. As far as relational theory is involved, the book [SS89, SS93] may be taken as the standard reference.

Heterogeneous decompositions are handled in Chapter 4, i.e. relations between possibly different sets, and we permute rows and columns independently. Contrasting hereto, we restrict in Chapter 5 to homogeneous relations on a set as well as to simultaneous permutations for visualization of rearranged relations. The same holds for Chapter 6, which is characterized by the fact that the decomposition is obtained by an iteration leading to some fixedpoint of a Galois correspondence. Termination and initial part, games, and assignments are studied. We also return to the homogeneous case of $n \times n$-relations on a set, however, with the attitude that the concepts of the preceding chapters are all available. This leads to difficult combinatorial situations to satisfy them. These are the well-known Frobenius-König and König-Egervary Theorems, here reformulated from the relation-algebraic side, not just in counting and cardinality style, thus giving deeper insight.

Chapter 7 shows the schema how theories may be extracted with these techniques from given relations. There is a relation given in the first place, which means that we have a theory (fragment) where we may talk on the domain and the range set and may formulate that R relates i and j. Every technique presented in Chapters 4 to 6 has as its result a decomposition of the row as well as the column set, that is unary predicates. It has in addition some problem-specific algebraic properties (an "ontology") between these subsets to be seen as theorems. We show schematically how the respective new and richer theory emerges therefrom, and may be used for querying, e.g.

[1] Be aware, that the first row of `rs` indicates that rows 1, 4, 6, 10, 12 are equal; it does not indicate — what one could easily misinterprete it to — that it gives example columns. Row 4 of `rs` shows the difference.

In an appendix further material is collected, e.g. how to pretty-print relations, and how to generate random relations for testing purposes. Help is also given for translating formulae into TEX, or for transferring a formula in component-free relational form to classical predicate-logic style.

Acknowledgement
Discussions with friends and colleagues from the inspiring TARSKI-environment during several COST management committee meetings are gratefully acknowledged. From the Institute for Software Technology and from the Informatics Faculty of the University of the German Armed Forces, Michael Ebert, Eric Offermann, and Michael Winter contributed to this study and discussed relational topics with me, thus providing considerable help.

2 Prerequisites and Tools

We prepare some tools. First we introduce Haskell as the programming language to be used. For this paper it is important to rearrange matrices according to some permutations so as to make their decomposition visible; so we present some programs to handle permutations. See also in the Appendix how to pretty-print relations or to generate relations randomly for testing purposes.

2.1 Haskell as Underlying Language

We use Haskell [HJW+92] as the programming language for this endeavour in explorative programming. It is a purely functional programming language, and it is widely accepted in research and university teaching. As Haskell is a referentially transparent programming language, it is well-suited for dealing with mathematical structures and to handle also logic-oriented and transformational tasks.

Some notations are self-explanatory — at least to all those versed in functional programming. For more information about Haskell see the Haskell WWW site at

URL: http://www.haskell.org/

(there you also find links to implementations), or the Journal of Functional Programming.

The present report is written in literate style as recommended mainly by Donald Knuth. This means that the ASCII source text of this TEX-document is at the same time an executable Haskell program. Working this way, one completely merges and integrates the written program with its documentation in TEX-style. The programs are strictly conformant to the Haskell 98 standard, and can therefore be expected to also be usable on future Haskell systems.

Certain graph-theoretic algorithms are expected to already exist, i.e., the Warshall algorithm and a shortest path algorithm.

2.2 Handling Permutations

In the example presented in the introduction, the resulting permutation has only been shown via the permuted row and column entries. The permutations, however, should be fully available in the program. There, they may be given as a function, decomposed into cycles, or as a permutation matrix:

$1 \mapsto 4$
$2 \mapsto 6$
$3 \mapsto 5$
$4 \mapsto 7$
$5 \mapsto 3$
$6 \mapsto 2$
$7 \mapsto 1$ or

```
[4,6,5,7,3,2,1]
```

```
[[1,4,7],[3,5],[6,2]]
```

$$\begin{pmatrix} 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Representing a permutation as a function, using cycles, or as a permutation matrix

Either form has its specific merits. Sometimes also the inverted permutation is useful. We therefore provide for types and functions to switch from one form to the other, and to apply a permutation to some list.

```
type PermMat = [[Bool]]
type PermFct = [Int]
type PermCyc = [[Int]]

applyPermFct :: Ord b => PermFct -> [b] -> [b]
applyPermFct p = map snd . sort . zip p

permFctToMat :: PermFct -> PermMat
permFctToMat p = map mkRow p    where mkRow i = take (length p) (map (i==) [1..])
```

It is possible to go back from a permutation matrix to the list form:

```
permMatToFct m =
    let zipWithNumbersRow r = zip r [1..]
        firstTruePosInRow r = snd $ head (dropWhile (\(a,b) -> not a)
                                          (zipWithNumbersRow r))
    in  map firstTruePosInRow m
```

Let us now convert back and forth to the cycle representation.

```
permFctToCyc :: PermFct -> PermCyc
permFctToCyc p =
     let len            = length p
         iteratePermFct = iterate (applyPermFct p) [1..len]
         allCycles      = transpose $ take len $ iteratePermFct
         normalizeCycle = cycleMinFst . nub
         cycleMinFst xs = take (length xs) $ dropWhile (/= minimum xs) (cycle xs)
     in  map reverse $ nub $ map normalizeCycle $ allCycles

permCycToFct :: PermCyc -> PermFct
permCycToFct =
  let cycleToPairs xs = take (length xs) $ zip (tail cycXs) cycXs
           where cycXs = cycle xs
      cyclesToPairs   = concatMap cycleToPairs
   in map snd . sort . cyclesToPairs . (map reverse)
```

Using `permFct` as the intermediate status, we manage to achieve other transitions.

```
permMatToCyc :: PermMat -> PermCyc
permMatToCyc = permFctToCyc . permMatToFct

permCycToMat :: PermCyc -> PermMat
permCycToMat = permFctToMat . permCycToFct
```

The following operations deliver the inverses of the preceding permutations in their respective form.

```
invPermFct :: PermFct -> PermFct
invPermFct = flip applyPermFct [1..]

invPermCyc :: PermCyc -> PermCyc
invPermCyc = map reverse

invPermMat :: PermMat -> PermMat
invPermMat = transpose
```

2.3 Permutations Determined by Partitions

Now permutations shall be determined from partitions. Let for example a list of partitioning subsets be given as `rs`:

```
   [[False,True, False,True, False,False],
    [False,False,True, False,False,True ],
    [True, False,False,False,True, False]]
```

We are interested to obtain a permutation like `[5,1,3,2,6,4]`, which puts `True`-entries of the first row before `True`-entries of the second row, etc. One may, according to an idea of Michael Ebert, rather easily obtain the inverted permutation function from a partition as follows: Annotate every element of the partition matrix with its column number and catenate all the rows thus annotated. This is then filtered according to whether the element is `True` or `False` and then stripped from its boolean value.

```
invsFctFromPart :: [[Bool]] -> PermFct
invsFctFromPart = let annotateColNum = map (zip [1..])
                      getAnnotation  = map fst
                      onlyTrueFields = filter snd
                  in  getAnnotation . onlyTrueFields . concat . annotateColNum
permFctFromPart :: [[Bool]] -> PermFct
permFctFromPart = invPermFct . invsFctFromPart
```

We may now also get the permutation from a partition together with its inverse as a matrix.

```
permMatFromPart rs = permFctToMat (permFctFromPart rs)
invsMatFromPart rs = permFctToMat (invsFctFromPart rs)
```

Using all these prerequisites, the rearrangement task is now easy.

```
rearrangeMatWithLines m rs cs =
     let rFT = invsFctFromPart rs
         rI  = permFctToMat    rFT
         rM  = invPermMat      rI
         cFT = invsFctFromPart cs
         cI  = permFctToMat    cFT
         cM  = invPermMat      cI
     in  (rI *** (m *** cM), rFT, cFT,
          linesPosition (rs *** rM),
          linesPosition (cs *** cM))
```

In `rearrangeMatWithLines` the rearranged matrix is returned together with additional structure, namely, the border lines separating zones of the matrix, and the row and column permutations in matrix form.

We provide in addition a version that brings better results for univalent and injective relations. Leaving rows fixed (except for sending null rows to the end) columns are arranged independently, also with null columns at the end. The aim is to make the 1:1-structure visible for a heterogeneous relation.

```
rearrangeDiagonal m =
   let nonemptyPartCol = filter or m
       nonemptyCol     = foldr1 (zipWith (||)) nonemptyPartCol
       emptyCol        = map not nonemptyCol
       nonemptyPartRow = filter or (transpMat m)
       nonemptyRow     = foldr1 (zipWith (||)) nonemptyPartRow
       emptyRow        = map not nonemptyRow
```

```
      rs = [nonemptyRow, emptyRow]
      cs = (nonemptyPartCol) ++ [emptyCol]
      (a,b,c,d,e) = rearrangeMatWithLines m rs cs
  in  stringForNamedMatrixLines (a,b,c,replicate (rows m) False, replicate (cols m) False)
```

A univalent and injective relation brought to diagonal style with zero rows and columns at the end

Such a result cannot be achieved for a homogeneous relation with the requirement to permute rows and columns simultaneously. This situation needs another treatment. The following permutes a univalent and injective relation to successor form. By this we mean: cycles to the front, then terminating linear strings, and finally null rows. Arrangement is such that the upper co-diagonal appears wherever possible.

```
permToSuccMatrix m =
   let trueFilter   = \(a,b) -> a == [True]
       trueNumbers  = \iv -> map snd (filter trueFilter (zip iv [1..]))
       initialVect  = negaMat (transpMat m *** (allMatNM (cols m) 1))
       initialNumb  = trueNumbers initialVect
       cyclVect     = negaMat $ transpMat (reflTranClosure m) *** initialVect
       cyclNumb     = trueNumbers cyclVect
       terminalVect = negaMat (m *** (allMatNM (cols m) 1))
       terminalNumb = trueNumbers terminalVect
       unRelated    = filter (   `elem` terminalNumb) initialNumb
       trulyInitial = filter (`notElem` terminalNumb) initialNumb
       followSequ p = snd (head (filter fst (zip (m !! (p-1)) [1..])))
       followCycle p ps q =
          let next = followSequ p
          in  if next == q then ps
                           else followCycle next (ps ++ [next]) q
       followString p ps =
          if p `elem` terminalNumb then (ps ++ [p])
                                   else followString (followSequ p) (ps ++ [p])
       allCycles1 [] = []
       allCycles1 (h:t) =
          let fs   = followCycle h [h] h
              rest = filter (\e -> e `notElem` fs) t
          in  fs : (allCycles1 rest)
       allCycles = allCycles1 cyclNumb
       allStrings1 [] = []
       allStrings1 (h:t) = [followString h []] ++ (allStrings1 t)
       allStrings   = allStrings1 trulyInitial
       occurringCyc = concat allCycles
       occurringStr = concat allStrings
       function     = occurringCyc ++ occurringStr ++ unRelated
   in  permFctToMat $ function
```

```
permToSuccForm m = p *** (m *** (transpMat p))    where p = permToSuccMatrix m
printSuccForm m =
   let permutation = permToSuccMatrix m
       permuted    = permToSuccForm   m
       difuClo     = difunctionalClosure (transpMat wI ||| wI)
                         where w  = warshall permuted
                               wI = w ||| (ident (rows w) &&& (w *** allMatFor w))
       rsH = nub (transpMat difuClo)
       rs  = if and (map or (transpMat rsH))
             then  reverse $ sort $ rsH
             else (reverse $ sort $ rsH) ++ [map not (map or (transpMat rsH))]
       original  = stringForNamedMatrixLines $
                   rearrangeMatWithLines (permutation *** (m *** (transpMat permutation)))
                                         [replicate (rows m) True] [replicate (rows m) True]
       origRearr = stringForNamedMatrixLines $ rearrangeMatWithLines permuted rs rs
   in  original ++ origRearr
```

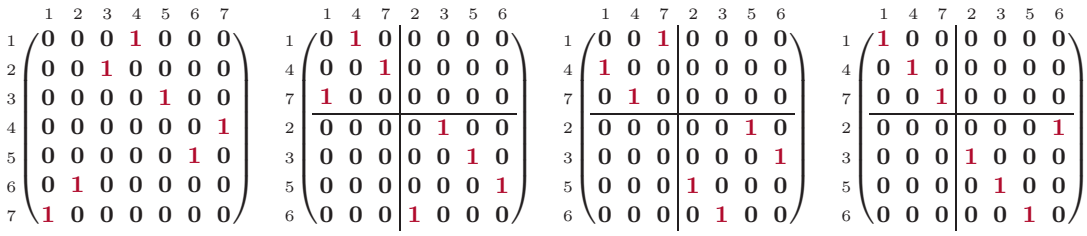
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
2	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
10	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
11	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
12	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
13	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
14	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
15	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
17	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
18	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
19	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	13	15	2	4	3	10	1	11	14	6	17	5	16	12	8	7	19	9	18
13	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
15	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
10	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
11	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
14	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
17	0	0	0	0	0	0	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	0	0	0	0	0	0
16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
12	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
19	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
18	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

A homogeneous univalent, and injective relation rearranged simultaneously

As an example we rearrange a permutation such that it decomposes into diagonal blocks of cyclic permutations.

```
permToBlockSuccessorForm f =
   let cs = permFctToCyc f
       p  = permFctToMat f
       cycPart z = foldr1 (\ c d -> zipWith (||) c d)
                     (map fst (filter (\(a,b) -> elem b z) (zip p [1..])))
       cycPartM  = map cycPart cs
       p2 = p *** p
       p3 = p *** p2
       rearr = stringForNamedMatrixLines $ rearrangeMatWithLines p  cycPartM cycPartM
       pow2  = stringForNamedMatrixLines $ rearrangeMatWithLines p2 cycPartM cycPartM
       pow3  = stringForNamedMatrixLines $ rearrangeMatWithLines p3 cycPartM cycPartM
   in  (stringForOriginalNamedMatrix p ++ rearr ++ pow2 ++ pow3)
```



Permutation and permutation rearranged to cycle form, and the latter raised to powers 2 and 3

From this form it is particularly easy to conclude that some power will be the identity, namely the least common multiple m of the block lengths. In addition, the transpose equals power $m - 1$.

3 Relation-Algebraic Preliminaries

Here we collect some well-known and often recalled material in order to make this report more or less self-contained. This time, we may restrict to relations as boolean matrices; therefore, some abstract concepts reduce to quite familiar ones.

3.1 Relation Algebra

When relations are studied on a moderately abstract level, they are conceived as subsets $R \subseteq X \times Y$ of the Cartesian product of the sets X, Y between which they are defined to hold. It is wise, however, to raise relational considerations to a more abstract level whenever possible. The formal definition of an abstract heterogeneous relation algebra — now widely agreed upon — is presented below.

3.1.1 Definition. A **heterogeneous relation algebra** is defined as a mathematical structure that

- is a category wrt. composition $;$ and identities $\mathbb{I}$,
- has complete atomic boolean lattices with $\cup, \cap, \overline{\ }, \mathbb{\bot}, \mathbb{T}, \subseteq$ as morphism sets,
- obeys rules for transposition in connection with the latter two that may be stated in either one of the following two ways:

  Dedekind: $R;S \cap Q \subseteq (R \cap Q;S^\mathsf{T});(S \cap R^\mathsf{T};Q)$ *or*
  Schröder: $A;B \subseteq C \iff A^\mathsf{T};\overline{C} \subseteq \overline{B} \iff \overline{C};B^\mathsf{T} \subseteq \overline{A}$ □

In order to avoid clumsy presentation, we shall adhere to the following policy in a heterogeneous algebra with its only partially defined operations: Whenever we say "For every X ...", we mean "For every X for which the construct in question is defined ...". We denote universal relations, e.g., with $\mathbb{T}$ which is imprecise, as we should also mention the objects between which the relation is meant to hold, $\mathbb{T}_{A,B}$.

The following is mainly recalled from [SS89, SS93], where also proofs may be found. In basic definitions, several equivalent variants are mentioned.

3.1.2 Proposition (*Row and column masks*)**.** The following formulas hold for arbitrary relations Q, R, S, provided the constructs are defined.

i) $$(Q \cap R;\mathbb{T});S = Q;S \cap R;\mathbb{T};$$

ii) $$(Q \cap (R;\mathbb{T})^\mathsf{T});S = Q;(S \cap R;\mathbb{T}).$$

3.1.3 Definition (*Defining properties of functions*)**.** If R is a relation, we call

$$\begin{array}{lll}
R \ \textbf{total} & :\Longleftrightarrow & \mathbb{T} = R;\mathbb{T} \iff \mathbb{I} \subseteq R;R^\mathsf{T} \iff \overline{R} \subseteq R;\overline{\mathbb{I}} \\
& \Longleftrightarrow & \text{For all } S, \text{ from } S;R = \mathbb{L} \text{ follows } S = \mathbb{L}; \\
R \ \textbf{univalent} & :\Longleftrightarrow & R^\mathsf{T};R \subseteq \mathbb{I} \iff R;\overline{\mathbb{I}} \subseteq \overline{R}; \\
R \ \textbf{surjective} & :\Longleftrightarrow & R^\mathsf{T} \text{ total} \\
& \Longleftrightarrow & \mathbb{T} = \mathbb{T};R \iff \mathbb{I} \subseteq R^\mathsf{T};R \iff \overline{R} \subseteq \overline{\mathbb{I}};R \\
& \Longleftrightarrow & \text{For all } S, \text{ from } R;S = \mathbb{L} \text{ follows } S = \mathbb{L}; \\
R \ \textbf{injective} & :\Longleftrightarrow & R^\mathsf{T} \text{ univalent} \\
& \Longleftrightarrow & R;R^\mathsf{T} \subseteq \mathbb{I} \iff \overline{\mathbb{I}};R \subseteq \overline{R}; \\
R \ \textbf{mapping} & :\Longleftrightarrow & R \text{ total and univalent} \iff R;\overline{\mathbb{I}} = \overline{R}.
\end{array}$$

□

The following laws hold for mappings and, more generally, for univalent relations.

3.1.4 Proposition (*Elementary rules for functions*)**.**

i)	Q, R univalent	$\Longrightarrow$	$Q;R$ univalent;
ii)	Q univalent	$\Longleftrightarrow$	$Q;(R \cap S) = Q;R \cap Q;S$ for all R, S;
iii)	Q univalent	$\Longleftrightarrow$	$R;Q \cap S = (R \cap S;Q^\mathsf{T});Q$ for all R, S;
iv)	$R \subseteq Q$, Q univalent; $R;\mathbb{T} \supseteq Q;\mathbb{T}$	$\Longrightarrow$	$R = Q$;
v)	Q univalent	$\Longleftrightarrow$	$Q;\overline{R} = Q;\mathbb{T} \cap \overline{Q;R}$ for all R.
vi)	Q mapping	$\Longleftrightarrow$	$Q;\overline{R} = \overline{Q;R}$ for all R.

While Schröder's rule always allows to transform a product on the smaller side of an inequality, very stringent conditions are necessary if the product occurs on the larger side.

3.1.5 Proposition. If F is a mapping, then arbitrary relations R, S, for which the constructs exist satisfy

$$R \subseteq S;F^\mathsf{T} \iff R;F \subseteq S.$$

Proof: Using $F^\mathsf{T};F \subseteq \mathbb{I}$ we have immediately "$\Longrightarrow$". For the other direction, we show $R = R;\mathbb{I} \subseteq R;F;F^\mathsf{T} \subseteq S;F^\mathsf{T}$. □

Another very basic fact is the so-called Tarski rule:

$$\mathbb{T};R;\mathbb{T} = \mathbb{T} \iff R \neq \mathbb{L};$$

for theoretical reasons, however, we will avoid using it as long as possible. Once used, the deduction will only be acceptable for a smaller class of relation algebras.

One sometimes asks, whether there exists the quotient of one relation with respect to another. The following concept of a **symmetric quotient** has very successfully been used in various application fields, not least in [BSZ86, BSZ90, BGS94]. We recall its definition

$$\mathtt{syq}(A, B) := \overline{A^\mathsf{T};\overline{B}} \cap \overline{A^\mathsf{T};B}.$$

and the basic algebraic rules:

3.1.6 Proposition (*Properties of the symmetric quotient*)**.**

i) $\mathtt{syq}(\overline{A},\overline{B}) = \mathtt{syq}(A,B)$;
ii) $\mathtt{syq}(B,A) = [\mathtt{syq}(A,B)]^\mathsf{T}$;
iii) $A ; \mathtt{syq}(A,A) = A$;
iv) $\mathbb{I} \subseteq \mathtt{syq}(A,A)$;
v) $\mathtt{syq}(A,B) \subseteq \mathtt{syq}(C;A, C;B)$ for every C;
vi) $F ; \mathtt{syq}(A,B) = \mathtt{syq}(A;F^\mathsf{T}, B)$ for every mapping F;
vii) $\mathtt{syq}(A,B) ; F = \mathtt{syq}(A, B;F)$ for every mapping F.

Always holds $A ; \mathtt{syq}(A,B) \subseteq B$ and we analyze in which way $A ; \mathtt{syq}(A,B)$ can differ from B. A column of $A ; \mathtt{syq}(A,B)$ is either equal to the corresponding column of B or it is zero. Having asked when $A ; \mathtt{syq}(A,B) = B$, a property characteristic of quotients, we now ask whether a cancellation rule holds for these quotients.

3.1.7 Proposition (*Cancelling of symmetric quotients*)**.** i) For arbitrary relations A, B, C we have

$$\mathtt{syq}(A,B) ; \mathtt{syq}(B,C) = \mathtt{syq}(A,C) \cap \mathtt{syq}(A,B) ; \mathbb{T} = \mathtt{syq}(A,C) \cap \mathbb{T} ; \mathtt{syq}(B,C).$$

ii) If $\mathtt{syq}(A,B)$ is total, or if $\mathtt{syq}(B,C)$ is surjective, then

$$\mathtt{syq}(A,B) ; \mathtt{syq}(B,C) = \mathtt{syq}(A,C).$$

In addition to these concepts, we will often use transitivity. A relation R is **transitive** if $R ; R \subseteq R$. It is well-known that every relation R has a **transitive closure** R^+ which is the least transitive relation containing it. With R^* we denote the reflexive-transitive closure of R. As is well-known, transitive relations are central for defining an ordering and an equivalence.

In graph theory, one is not restricted to study homogeneous relations, i.e., graphs on a point set. Often directed graphs or hypergraphs are investigated. In these cases, one has heterogeneous relations saying which arc or hyperedge is incident with which point or vertex of the graph. Then one is accustomed to use the concept of an **edge-adjacency** $\mathrm{K} := \overline{\mathbb{I}} \cap M ; M^\mathsf{T}$. Two distinct arcs or hyperedges are called adjacent if there is a vertex that is incident with both. So adjacency can be expressed by the product of the incidence and its transpose.

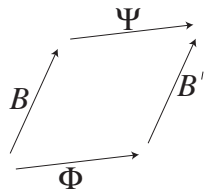
3.2 Basic Relational Operators in Haskell

We will here 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. The functions `allMatNM` and `nullMatNM` need row and column numbers as arguments first and will then generate universal and null matrices in the same way as `ident` will provide an identity relation after giving it a row number. There is a way of generating boolean null matrices of a given size by `nullMatFor` if the shape is provided by some other matrix, analogously for `allMatFor` and `lidentFor, rIdentFor`. Relations will be transposed by `transpMat` and negated by `negaMat`.

There is also the effect, that denoting a row, e.g., can be done by giving its number, a boolean vector indicating the position, or a 1-column boolean matrix. At least in the course of work, these forms occur. There are some basic functions switching from one version to the other, namely `pointNumToVec, pointNumToMat, pointVecToNum`, etc.

3.3 Congruences and Coverings

We recall the concept of homomorphism for relational structures explaining it for unary ones. Structure and mappings shall commmute, however, not as equality but just as containment.



Relational homomorphism

3.3.1 Definition. Given two relations B, B', we call the pair (Φ, Ψ) of relations a **homomorphism** from B to B', if

$$\Phi^\mathsf{T};\Phi \subseteq \mathbb{I}, \quad \mathbb{I} \subseteq \Phi;\Phi^\mathsf{T}, \quad \Psi^\mathsf{T};\Psi \subseteq \mathbb{I}, \quad \mathbb{I} \subseteq \Psi;\Psi^\mathsf{T}, \qquad B;\Psi \subseteq \Phi;B'$$

i.e. if Φ, Ψ are mappings satisfying $B;\Psi \subseteq \Phi;B'$. □

The homomorphism condition proper has four variants which may be used interchangeably:

3.3.2 Proposition. If Φ, Ψ are mappings, then

$$B;\Psi \subseteq \Phi;B' \iff B \subseteq \Phi;B';\Psi^\mathsf{T} \iff \Phi^\mathsf{T};B \subseteq B';\Psi^\mathsf{T} \iff \Phi^\mathsf{T};B;\Psi \subseteq B'$$ □

As usual, also isomorphisms are introduced.

3.3.3 Definition. We call (Φ, Ψ) an **isomorphism** between the two relations B, B', if it is a homomorphism from B to B' and $(\Phi^\mathsf{T}, \Psi^\mathsf{T})$ is a homomorphism from B' to B. □

We should mention a bit of theory to end this paragraph. Whenever some equivalence behaves well with regard to some other structure, we are accustomed to call it a congruence. This is well-known for algebraic structures, i.e., those defined by mappings on some set. We define it correspondingly for the non-algebraic case, including heterogeneous relations. While the basic idea is known from many application fields, the following general concepts may be a new abstraction.

3.3.4 Definition. Let B be a relation and Ξ, Θ equivalences. The pair (Ξ, Θ) is called a **B-congruence** if $\Xi;B \subseteq B;\Theta$. □

If B were a binary operation on a given set and we had $\Xi = \Theta$, we would say that B "has the substitution property with regard to Ξ". The concept of congruence is related to the concept of a multi-covering.

3.3.5 Definition. A homomorphism (Φ, Ψ) from B to B' is called a **multi-covering**, provided the functions are surjective and satisfy $\Phi;B' \subseteq B;\Psi$ in addition to being a homomorphism. □

The relationship between congruences and multi-coverings is very close.

3.3.6 Theorem.

i) If (Φ, Ψ) is a multi-covering from B to B', then $(\Xi, \Theta) := (\Phi;\Phi^\mathsf{T}, \Psi;\Psi^\mathsf{T})$ is a B-congruence.

ii) If the pair (Ξ, Θ) is a B-congruence, then there exists up to isomorphism at most one multi-covering (Φ, Ψ) satisfying $\Xi = \Phi;\Phi^\mathsf{T}$ and $\Theta = \Psi;\Psi^\mathsf{T}$.

Proof: i) Ξ is certainly reflexive and transitive, as Φ is total and univalent. In the same way, Θ is reflexive and transitive. The relation $\Xi = \Phi;\Phi^\mathsf{T}$ is symmetric by construction and so is Θ. Now we prove $\Xi;B = \Phi;\Phi^\mathsf{T};B \subseteq \Phi;B';\Psi^\mathsf{T} \subseteq B;\Psi;\Psi^\mathsf{T} = B;\Theta$ applying one after the other the definition of Ξ, one of the homomorphism definitions, the multi-covering condition, and the definition of Θ.

ii) Let (Φ_i, Ψ_i) be a multi-covering from B to B_i, $i = 1, 2$. Then $B_i \subseteq \Phi_i^\mathsf{T};\Phi_i;B_i \subseteq \Phi_i^\mathsf{T};B;\Psi_i \subseteq B_i$, and therefore "=", applying surjectivity, the multi-covering property and one of the homomorphism conditions.

Now we show that $(\xi, \theta) := (\Phi_1^\mathsf{T};\Phi_2, \Psi_1^\mathsf{T};\Psi_2)$ is a homomorphism from B_1 onto B_2 — which is then of course also an isomorphism.

$\xi^\mathsf{T};\xi = \Phi_2^\mathsf{T};\Phi_1;\Phi_1^\mathsf{T};\Phi_2 = \Phi_2^\mathsf{T};\Xi;\Phi_2 = \Phi_2^\mathsf{T};\Phi_2;\Phi_2^\mathsf{T};\Phi_2 = \mathbb{I};\mathbb{I} = \mathbb{I}$

$B_1;\theta = \Phi_1^\mathsf{T};B;\Psi_1;\Psi_1^\mathsf{T};\Psi_2 = \Phi_1^\mathsf{T};B;\Theta;\Psi_2 = \Phi_1^\mathsf{T};B;\Psi_2;\Psi_2^\mathsf{T};\Psi_2 \subseteq \Phi_1^\mathsf{T};\Phi_2;B_2;\mathbb{I} = \xi;B_2$ □

Having these results in mind, we are in a position to prove generalisations of Birkhoffs famous theorem on the lattice of congruences for algebraic structures extending it to relational structures. Remember, that $R^\mathsf{T};R \subseteq \mathbb{I}$ (univalency) will always hold for algebraic structures.

3.3.7 Theorem. Let some finite heterogeneous relation R be given. Then all R-congruences (P, Q) satisfying $R^\mathsf{T};R \subseteq Q$ form a complete lattice with least element $(\mathbb{I}, \Theta)$, where $\Theta := (R^\mathsf{T};R)^*$.

Proof: Intersections of equivalences are equivalences again and the congruences considered behave $\cap$-hereditary:

$$\begin{aligned}(P_1 \cap P_2);R &\subseteq P_1;R \cap P_2;R \subseteq R;Q_1 \cap R;Q_2 \subseteq (R \cap R;Q_2;Q_1^\mathsf{T});(Q_1 \cap R^\mathsf{T};R;Q_2)\\ &\subseteq R;(Q_1 \cap Q_2;Q_2) = R;(Q_1 \cap Q_2)\end{aligned}$$

Unions of equivalences usually fail to be equivalences, but the transitive closure of a union is an equivalence. For this, we use the well-known formula $(P_1 \cup P_2)^* = (P_1;P_2)^*P_1^*$ from regular algebra to show

$$(P_1 \cup P_2)^*;R = (P_1;P_2)^*;P_1;R \subseteq (P_1;P_2)^*;R;Q_1 \subseteq R;(Q_1;Q_2)^*;Q_1 \subseteq R;(Q_1 \cup Q_2)^*$$

So a complete lattice is established for the finite case handled here, and we consider its least element. Obviously, $\mathbb{I}$ and $\Theta := (R^\mathsf{T};R)^*$, are R-congruences as they are both equivalences, and $\mathbb{I};R = R \subseteq R;(R^\mathsf{T};R)^* = R;\Theta$. A smaller congruence satisfying $R^\mathsf{T};R \subseteq Q$ can obviously not be found. □

The multi-covering (Φ, Ψ) for some given congruences P, Q need not exist in the given relation algebra. It may, however, be constructed by setting Φ, Ψ to be the quotient mappings according to the equivalences P, Q together with $R' := \Phi^\mathsf{T};R;\Psi$, in which case we get R' univalent.

3.3.8 Theorem. Let some finite heterogeneous relation R be given. Then all R-congruences (P, Q) which satisfy both, $R;R^\mathsf{T} \subseteq P$ and $R^\mathsf{T};R \subseteq Q$, form a complete lattice, the least element of which is $(\Xi, \Theta) := ((R;R^\mathsf{T})^*, (R^\mathsf{T};R)^*)$.

Proof: The proof follows the same schema as the previous one. Again the set
$\{(P,Q) \mid (P,Q)$ are R-congruences satisfying $R ; R^\mathsf{T} \subseteq P$ and $R^\mathsf{T} ; R \subseteq Q\}$
is $\cap$-hereditary and smaller congruences cannot be found. □

The multi-covering (Φ, Ψ) onto R' will this time result in a univalent and also injective relation R'. Should R happen to be total and surjective, R' would be a bijective mapping between two sets; see also the forthcoming Def. 4.1.6.

There is another point to mention here which has gained considerable interest in an algebraic or topological context, not least for Riemann surfaces.

3.3.9 Proposition (*Lifting property*)**.** Let a homogeneous relation B be given together with a multi-covering Φ on the relation B'. Let furthermore some rooted graph B_0, i.e., satisfying $B_0 ; B_0^\mathsf{T} \subseteq \mathbb{I}$ and $B_0^{\mathsf{T}*} ; a_0 = \mathbb{T}$, be given together with a homomorphism Φ_0 that sends the root a_0 to $a' := \Phi_0^\mathsf{T} ; a_0$. If $a \subseteq \Phi^\mathsf{T} ; a'$ is some point mapped by the multi-covering Φ to a', there exists always a relation Ψ — not necessarily a mapping — satisfying the properties
$\Psi^\mathsf{T} ; a_0 = a$ and $B_0 ; \Psi \subseteq \Psi ; B$.

Proof: Define recursively $\Psi := \mathtt{inf}\{X \mid a_0 ; a^\mathsf{T} \cup (B_0^\mathsf{T} ; X ; B \cap \Phi_0 ; \Phi^\mathsf{T}) \subseteq X\}$ □

The relation enjoys the homomorphism property but fails to be a mapping in general. In order to make it a mapping, one has to choose one of the following two possibilities:

- Firstly, one might follow the recursive definition starting from a_0 and at every stage make an arbitrary choice among the relational images offered, thus choosing a fiber.
- Secondly, one may further restrict the multi-covering condition to "locally univalent" fans in Φ, requiring $B_0^\mathsf{T} ; \Psi ; B \cap \Phi_0 ; \Phi^\mathsf{T} \subseteq \mathbb{I}$ to hold for it, which leads to a well-developed theory.

In both cases, one will find a homomorphism from B_0 to B.

3.4 Properties of Idempotent Relations

We insert here some remarks applying to idempotent relations.

3.4.1 Proposition. For an idempotent boolean matrix A, $a_{kk} = \mathbf{0}$ implies that column k may be represented as a union of the other columns. This holds correspondingly for rows. Furthermore, the minor corresponding row a_{kk} is again idempotent.

Proof: Without loss of generality, we may assume $k = 1$ confronting us with $A = \begin{pmatrix} \mathbf{0} & u^\mathsf{T} \\ v & W \end{pmatrix}$, where u, v are vectors and W is a boolean matrix. From $A^2 = A$ one deduces easily that

$$u^\mathsf{T} ; v = \mathbf{0}\,, \quad u^\mathsf{T} ; W = u^\mathsf{T}, \quad W ; v = v, \quad v ; u^\mathsf{T} \cup W^2 = W.$$

Therefore, the first row — resp. the first column — may be represented as

$$\begin{pmatrix} \mathbf{0} \\ v \end{pmatrix} = \begin{pmatrix} u^\mathsf{T} \\ W \end{pmatrix} ; v, \qquad (\ \mathbf{0} \quad u^\mathsf{T}\ ) = u^\mathsf{T} ; (\ v \quad W\ )\,.$$

We show $W^2 = W$ observing that $W;v = v$ implies $W^2 = W;(v;u^\mathsf{T} \cup W^2) \supseteq W;v;u^\mathsf{T} = v;u^\mathsf{T}$. □

3.4.2 Proposition. For an idempotent boolean matrix A with $\mathbb{I} \subseteq A$, column k may be represented as a union of the other columns precisely when this holds correspondingly for row k. Furthermore, the minor corresponding row a_{kk} is always idempotent.

Proof: Without loss of generality, we may assume $k = 1$ confronting us with $A = \begin{pmatrix} \mathbf{1} & u^\mathsf{T} \\ v & W \end{pmatrix}$, where u, v are vectors and W is a boolean matrix. From $A^2 = A$ one deduces easily that

$$u^\mathsf{T} \cup u^\mathsf{T};W = u^\mathsf{T}, \quad v \cup W;v = W;v = v, \quad v;u^\mathsf{T} \cup W^2 = W.$$

The middle equation has already been sharpened a bit observing that $\mathbb{I} \subseteq W$.

Assume that $(\ \mathbf{1} \quad u^\mathsf{T}\ )$ is representable as a union of some other rows, which means that with some vector x

$$(\ \mathbf{1} \quad u^\mathsf{T}\ ) = x^\mathsf{T};(\ v \quad W\ ).$$

Then $x^\mathsf{T};v = \mathbf{1}$ and $x^\mathsf{T};W = u^\mathsf{T}$. Now, $u^\mathsf{T};v = x^\mathsf{T};W;v = x^\mathsf{T};v = \mathbf{1}$, so that

$$\begin{pmatrix} \mathbf{1} \\ v \end{pmatrix} = \begin{pmatrix} u^\mathsf{T} \\ W \end{pmatrix};v.$$

We show $W^2 = W$ observing $W^2 \subseteq W$ from above and $W \supseteq \mathbb{I}$ as assumed. □

4 Heterogeneous Decompositions

Two different cases will be distinguished:

- *Heterogeneous* relations between two sets, for which we will permute rows and columns *independently* — be it that the sets have equal cardinality.
- *Homogeneous* relations in which rows and columns have to be rearranged *simultaneously*.

We begin with the more general case of heterogeneous relations and handle decomposition of matrices, either to some sort of block diagonal decomposition (the difunctional case) or to upper block triangular form (the Ferrer's case). Covering all $\mathbf{1}$-entries of a boolean matrix by as few lines (i.e., rows or columns) as possible will be shown to correspond to the task of finding as many lines (again rows or columns) possible independent from one another.

4.1 Difunctional Relations

Difunctionality leads to quite an important decomposition. It groups rows as well as columns so as to obtain a partial 1:1-correspondence of classes. After rearranging, this means that a "partial block diagonal" is obtained. Some of this material has already been presented as an example in the introduction.

4.1.1 Definition. i) A relation R is called **difunctional**[1] if $R ; R^\mathsf{T} ; R = R$.

ii) For every relation R, the least difunctional relation containing it is well-defined and we define (according to J. Riguet) the **difunctional closure** as

$$h_{\mathrm{difu}}(R) := \mathtt{inf}\,\{H \mid R \subseteq H \text{ with } H \text{ difunctional}\} \qquad \square$$

We recall the proof that besides the descriptive version just given there is also a constructive definition of this closure, which we will afterwards use for computing it.

4.1.2 Proposition. $h_{\mathrm{difu}}(R) = R ; (R^\mathsf{T} ; R)^+ = (R ; R^\mathsf{T})^+ ; R = (R ; R^\mathsf{T})^+ ; R ; (R^\mathsf{T} ; R)^+ = R ; (R^\mathsf{T} ; R)^*$.

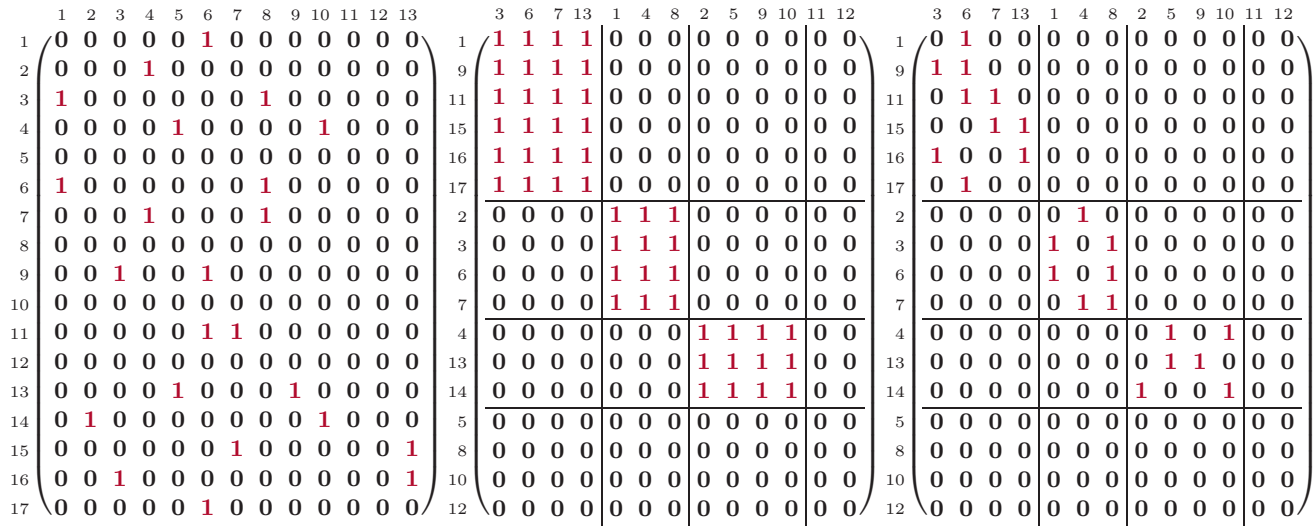
Proof: h_{difu} is a closure operation because the property of being difunctional is easily shown to be $\cap$-hereditary. Difunctionality of $R ; (R^\mathsf{T} ; R)^+$ is rather trivial, so that $D := h_{\mathrm{difu}}(R) \subseteq R ; (R^\mathsf{T} ; R)^+$. Conversely with $R \subseteq D$ also $D ; R^\mathsf{T} ; R \subseteq D ; D^\mathsf{T} ; D \subseteq D$ hold, since D is difunctional. Therefore, iteratively applied, $R ; (R^\mathsf{T} ; R)^+ \subseteq D$. $\square$

[1] Demanding " $\subseteq$ " to hold would suffice, as " $\supseteq$ " is satisfied for all relations.

```
difunctionalClosure r = r *** (warshall (transpMat r *** r))
difunctionalTest    r = difunctionalClosure r <== r
```

In our example from the introduction we have presented a difunctional matrix in its original and in its rearranged form. The effect was rather obvious. In the next example, we rearrange a non-closed matrix according to its difunctional closure.

```
printResDifuClosure m =
   let difuClo = difunctionalClosure m
       rsH = nub (transpMat difuClo)
       rs  = if and (map or (transpMat rsH))
             then  reverse $ sort $ rsH
             else (reverse $ sort $ rsH) ++ [map not (map or (transpMat rsH))]
       csH = nub difuClo
       cs  = if and (map or (transpMat csH))
             then csH
             else csH ++ [map not (map or (transpMat csH))]
       origRearr = stringForNamedMatrixLines $ rearrangeMatWithLines m       rs cs
       closRearr = stringForNamedMatrixLines $ rearrangeMatWithLines difuClo rs cs
   in  ((stringForOriginalNamedMatrix m) ++ closRearr ++ origRearr)
```



A relation with its difunctional closure and in an accordingly rearranged form

Related questions sometimes come up in cluster analysis. Much of the information a difunctional relation gives us, is already present in its reduced form. One should, however, know how the rows and columns relate to the reduced matrix. This is what `difuFactorization` provides.

```
difuFactorization r =
   let rowWork  (f, r) z =
          let compare = map (\s -> z == s) r
          in  case (or compare) of
                 True  -> (f ++ [compare],           r)
                 False -> (transpMat ((transpMat f)
                                  ++ [take (rows f) (repeat False)])
                                  ++ [compare ++ [True]], r ++ [z])
       difuWork (f, r) []         = (f, r)
       difuWork (f, r) (z : rest) = difuWork (rowWork (f, r) z) rest
       rRowReduced = difuWork ([], []) r
       f           = fst rRowReduced
```

```
      rColReduced = difuWork ([], []) (transpMat (snd rRowReduced))
      g           = fst rColReduced
      rRed        = transpMat (snd rColReduced)
in    (f, rRed, g)
```

The resulting relation has, however, a much more economic way of representing it. The result gives the difunctional toy argument matrix factorized $R = F ; R_{\text{condensed}} ; G^\mathsf{T}$ into the product of a mapping, the reduced matrix, and a transposed mapping.

$$\left(\begin{array}{cccccccc|cccc|cc}
1&1&1&1&1&1&1&1&0&0&0&0&0&0\\
1&1&1&1&1&1&1&1&0&0&0&0&0&0\\
1&1&1&1&1&1&1&1&0&0&0&0&0&0\\
1&1&1&1&1&1&1&1&0&0&0&0&0&0\\
1&1&1&1&1&1&1&1&0&0&0&0&0&0\\
\hline
0&0&0&0&0&0&0&0&1&1&1&1&0&0\\
0&0&0&0&0&0&0&0&1&1&1&1&0&0\\
0&0&0&0&0&0&0&0&1&1&1&1&0&0\\
\hline
0&0&0&0&0&0&0&0&0&0&0&0&1&1\\
0&0&0&0&0&0&0&0&0&0&0&0&1&1\\
0&0&0&0&0&0&0&0&0&0&0&0&1&1\\
\hline
0&0&0&0&0&0&0&0&0&0&0&0&0&0
\end{array}\right)
=
\left(\begin{array}{cccc}
1&0&0&0\\
1&0&0&0\\
1&0&0&0\\
1&0&0&0\\
1&0&0&0\\
\hline
0&1&0&0\\
0&1&0&0\\
0&1&0&0\\
\hline
0&0&1&0\\
0&0&1&0\\
0&0&1&0\\
\hline
0&0&0&1
\end{array}\right)
;
\left(\begin{array}{ccc}
1&0&0\\
0&1&0\\
0&0&1\\
0&0&0
\end{array}\right)
;
\left(\begin{array}{ccc}
1&0&0\\
1&0&0\\
1&0&0\\
1&0&0\\
1&0&0\\
1&0&0\\
1&0&0\\
1&0&0\\
\hline
0&1&0\\
0&1&0\\
0&1&0\\
0&1&0\\
\hline
0&0&1\\
0&0&1
\end{array}\right)^\mathsf{T}$$

Representing a difunctional relation by two functions and a smaller relation

Meanwhile, we have some feeling how a difunctional matrix looks like. We also know the algebraic characterization from the definition. Now we ask for the practical aspects of this definition, which has long been discussed and is known among numerical analysts as chainability.

4.1.3 Definition. Let a relation R be given that is conceived as a chessboard with dark squares or white according to whether R_{ik} is `True` or `False`. A rook shall operate on the chessboard in horizontal or vertical direction; however, it is only allowed to change direction on dark squares. Using this interpretation, a relation R is called **chainable** if the non-vanishing entries (i.e., the dark squares) can all be reached from one another by a sequence of "rook moves", or else if $h_{\text{difu}}(R) = \mathbb{T}$. □

```
chainableTest r = difunctionalClosure r === allMatFor r
```

We illustrate this definition mentioning a related concept. The relation R shall for the moment be conceived as a hypergraph-incidence between hyperedges (rows) and vertices (columns). Then $K := \overline{\mathbb{I}} \cap R ; R^\mathsf{T}$ is the so-called edge-adjacency, see e.g. [SS89, SS93].

4.1.4 Proposition. A total and surjective relation R is chainable precisely when its edge-adjacency K is strongly connected.

Proof: First we show that $K^* = (R ; R^\mathsf{T})^*$, using the formula $(A \cup B)^* = (A^* ; B)^* ; A^*$, well-known from regular algebra. Then

$$(R ; R^\mathsf{T})^* = ((\mathbb{I} \cap R ; R^\mathsf{T}) \cup (\overline{\mathbb{I}} \cap R ; R^\mathsf{T}))^* = ((\mathbb{I} \cap R ; R^\mathsf{T})^* ; (\overline{\mathbb{I}} \cap R ; R^\mathsf{T}))^* ; (\mathbb{I} \cap R ; R^\mathsf{T})^* = (\mathbb{I} ; K)^* ; \mathbb{I} = K^*$$

From chainability, $h_{\text{difu}}(R) = (R ; R^\mathsf{T})^* ; R = \mathbb{T}$ we get immediately $K^* = (R ; R^\mathsf{T})^* \supseteq (R ; R^\mathsf{T})^+ = (R ; R^\mathsf{T})^* ; R ; R^\mathsf{T} = \mathbb{T} ; R^\mathsf{T}$. Totality of R gives the first direction; it is necessary, since there might exist an empty hyperedge completely unrelated to the vertices.

The other direction is $h_{\text{difu}}(R) = (R;R^\intercal)^*;R = K^*;R = \top;R$, where R must be surjective, as otherwise there might exist an isolated vertex unrelated to all the edges. □

At this point, we resume our investigation of congruences from Props. 3.3.7 and 3.3.8, relating them to difunctionality.

4.1.5 Proposition. Let some possibly heterogeneous relation R be given and consider the constructs $\Xi := (R;R^\intercal)^*$ and $\Xi' := (R^\intercal;R)^*$.

i) Ξ and Ξ' are equivalences.

ii) The pair (Ξ, Ξ') forms an R-congruence, i.e. $\Xi;R \subseteq R;\Xi'$; in addition $\Xi;R = R;\Xi'$.

iii) The pair (Ξ', Ξ) forms an $R^\intercal$-congruence, i.e. $\Xi';R^\intercal \subseteq R^\intercal;\Xi$; in addition $\Xi';R^\intercal = R^\intercal;\Xi$.

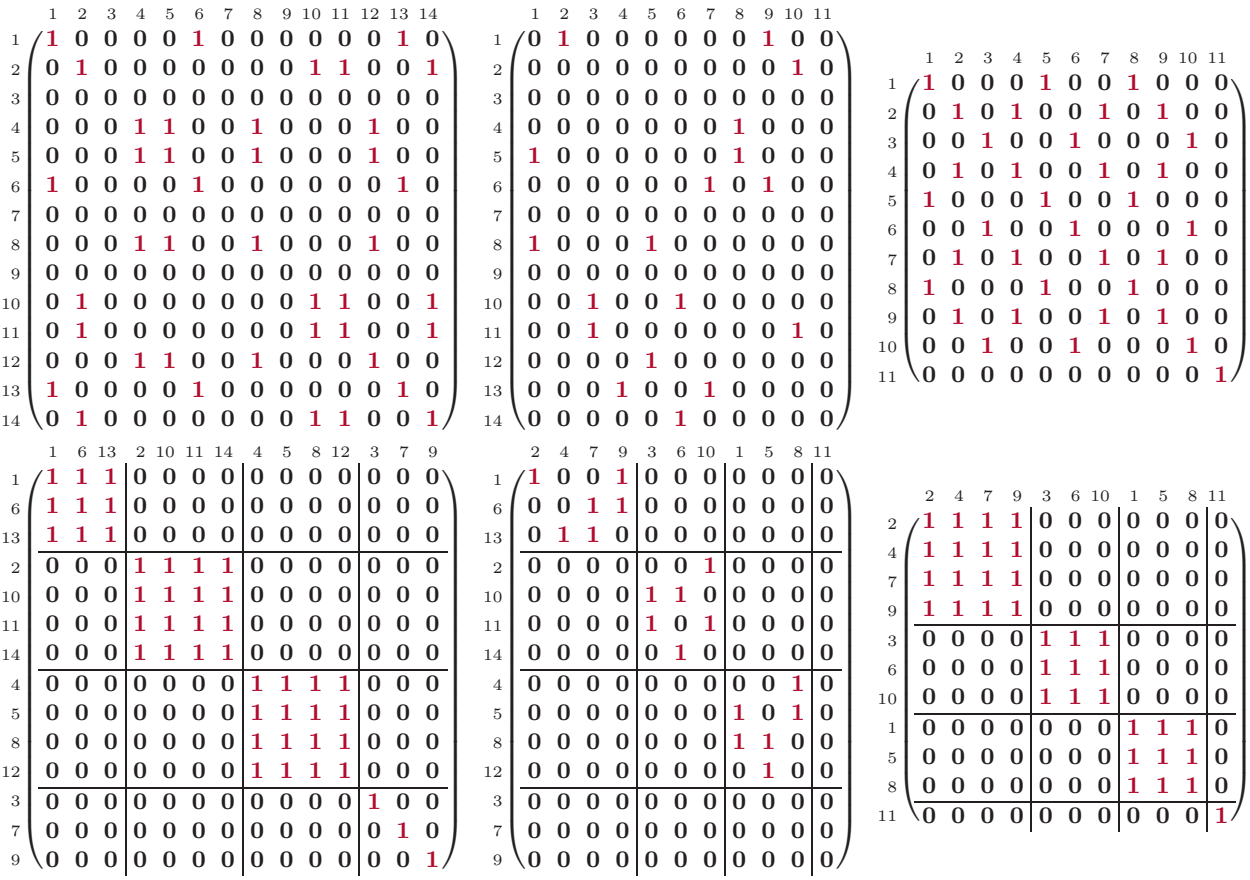
iv) "Considered modulo Ξ, Ξ'", the relation R is univalent $R^\intercal;\Xi;R \subseteq \Xi'$ and injective $R;\Xi';R^\intercal \subseteq \Xi$.

v) $h_{\text{difu}}(R) = \Xi;R;\Xi'$.

Proof: The proofs follow easily from regular algebra. □

As a result, there is a partial 1:1-correspondence between the classes according to the two equivalence relations Ξ, Ξ' on the domain and on the range side, respectively. One might wish to call it the natural congruence of R. The following function gives the "arbitrary" relation in the middle and the two congruences accordingly on the left and on the right.

```
printResCongrMatrix m =
   let mT   = transpMat m
       mmT  = reflTranClosure (m  *** mT)
       mTm  = reflTranClosure (mT *** m )
       difuClo = difunctionalClosure m
       rsH = nub (transpMat difuClo)
       rs  = if and (map or (transpMat rsH))
             then  reverse $ sort $ rsH
             else (reverse $ sort $ rsH) ++ [map not (map or (transpMat rsH))]
       csH = nub difuClo
       cs  = if and (map or (transpMat csH))
             then csH
             else csH ++ [map not (map or (transpMat csH))]
       origRearr  = stringForNamedMatrixLines $ rearrangeMatWithLines m   rs cs
       leftCongr  = stringForNamedMatrixLines $ rearrangeMatWithLines mmT rs rs
       rightCongr = stringForNamedMatrixLines $ rearrangeMatWithLines mTm cs cs
   in   ((stringForOriginalNamedMatrix m  ) ++ (stringForOriginalNamedMatrix mmT) ++
         (stringForOriginalNamedMatrix mTm) ++ leftCongr ++ origRearr ++ rightCongr)
```



Relation decomposed according to its natural congruence and the same in rearranged form

The following concepts of inverses are known in the context of linear algebra for numerical problems.

4.1.6 Definition. Let some relation A be given. The relation G is called

i) a **generalized inverse** of A if $A;G;A = A$.

ii) a **Moore-Penrose inverse** of A if the following four conditions hold
$A;G;A = A, \qquad G;A;G = G, \qquad (A;G)^\mathsf{T} = A;G, \qquad (G;A)^\mathsf{T} = G;A.$

We recall a well-known result on these Moore-Penrose inverses.

4.1.7 Theorem. Moore-Penrose inverses are uniquely determined provided they exist.

Proof: Assume two Moore-Penrose inverses G, H of A to be given. Then we may proceed as follows $G = G;A;G = G;G^\mathsf{T};A^\mathsf{T} = G;G^\mathsf{T};A^\mathsf{T};H^\mathsf{T};A^\mathsf{T} = G;G^\mathsf{T};A^\mathsf{T};A;H = G;A;G;A;H = G;A;H = G;A;H;A;H = G;A;A^\mathsf{T};H^\mathsf{T};H = A^\mathsf{T};G^\mathsf{T};A^\mathsf{T};H^\mathsf{T};H = A^\mathsf{T};H^\mathsf{T};H = H;A;H = H.$ □

We now relate these concepts with permutations and difunctionality.

4.1.8 Theorem. For a relation A, the following are equivalent:

i) A has a Moore-Penrose inverse.

ii) A has A^T as its Moore-Penrose inverse.

iii) A is difunctional.

iv) Any two rows (or columns) of A are either disjoint or identical.

v) There exist permutation matrices P, Q such that $P;A;Q$ has block-diagonal form, i.e.

$$P;A;Q = \begin{pmatrix} B_1 & \amalg & \amalg & \amalg & \amalg & \amalg \\ \amalg & B_2 & \amalg & \amalg & \amalg & \amalg \\ \amalg & \amalg & B_3 & \amalg & \amalg & \amalg \\ \amalg & \amalg & \amalg & B_4 & \amalg & \amalg \\ \amalg & \amalg & \amalg & \amalg & B_5 & \amalg \\ \amalg & \amalg & \amalg & \amalg & \amalg & \amalg \end{pmatrix}$$

with not necessarily square diagonal entries $B_i = \top\!\!\top$. □

4.2 Relations of Ferrer's Type

The following decomposition is applicable only in rather rare cases. It may be handled similarly and groups rows as well as columns so as to obtain a "linear ordering" of classes. After rearranging, the resulting matrix will have upper triangular form — up to the fact that this is meant blockwisely.

4.2.1 Definition. We say that a relation

$$\begin{aligned} A \text{ is of } \textbf{Ferrer's type} \quad :\Longleftrightarrow \quad & A;\overline{A}^\mathsf{T};A \subseteq A \iff \overline{A};A^\mathsf{T};\overline{A} \subseteq \overline{A} \iff A^\mathsf{T};\overline{A};A^\mathsf{T} \subseteq A^\mathsf{T} \\ \Longleftrightarrow \quad & A \text{ can be written in staircase block form} \\ & \text{by suitably rearranging rows and columns.} \end{aligned}$$

□

```
ferrerOperator r = r *** (negaMat (transpMat r)) *** r
ferrersTest    r = ferrerOperator r <== r
ferrerInterior r = transpMat (negaMat (ferrerOperator r))
ferrerTransTest r = r *** (ferrerInterior r *** r)
```

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	0	1	0	0	0	1	0	0	0	1	0	0	1	0	1
2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0
4	1	1	0	0	1	1	1	1	1	1	1	1	1	1	1
5	0	1	0	0	0	1	0	0	0	1	0	0	1	0	1
6	0	1	0	0	0	1	0	0	0	1	0	0	1	0	1
7	0	1	0	0	0	1	0	0	0	1	0	0	1	0	1
8	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0
9	1	1	0	0	1	1	1	1	1	1	1	1	1	1	1
10	0	1	0	0	0	1	0	0	0	1	0	0	1	0	1
11	0	1	0	0	0	1	1	0	1	1	0	1	1	0	1
12	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0
13	0	1	0	0	0	1	1	0	1	1	0	1	1	0	1
14	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
15	0	1	0	0	0	1	1	0	1	1	0	1	1	0	1
16	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0
17	0	1	0	0	0	1	0	0	0	1	0	0	1	0	1
18	0	1	0	0	0	1	1	0	1	1	0	1	1	0	1

	3	4	1	5	8	11	14	7	9	12	2	10	15	6	13
4	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1
9	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1
11	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
13	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
15	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
18	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
1	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1
5	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1
6	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1
7	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1
10	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1
17	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1
3	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1
8	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1
12	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1
16	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
14	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Original relation of Ferrer's type and rearranged version

```
printResFerrMatrix m =
   let exstinguish []    = []
       exstinguish (h:t) = h : (map (\z -> zipWith (&&) (map not h) z) (exstinguish t))
       rsH  = nub (transpMat m)
       rsH1 = exstinguish $ sort rsH
       rs   = if and (map or (transpMat rsH1))
                     then rsH1
                     else (tail rsH1) ++
                          [map not (map or (transpMat rsH1))]
       csH = sort $ nub m
       csH1 = reverse (exstinguish csH)
       cs  = if and (map or (transpMat csH1))
                    then csH1
                    else [map not (map or (transpMat csH1))] ++ csH1
       ferrRearr = stringForNamedMatrixLines $ rearrangeMatWithLines m rs cs
   in  putStr ((stringForOriginalNamedMatrix m) ++ ferrRearr)

ferrerFactorization r =
   let rowWork (f, r) z =
          let compare = map (\s -> z == s) r
          in  case (or compare) of
                 True  -> (f ++ [compare],           r)
                 False -> (transpMat ((transpMat f)
                                  ++ [take (rows f) (repeat False)])
                                  ++ [compare ++ [True]], r ++ [z])
       ferrerWork (f, r) []         = (f, r)
       ferrerWork (f, r) (z : rest) = ferrerWork (rowWork (f, r) z) rest
       rRowReduced = ferrerWork ([], []) r
       f           = fst rRowReduced
       rColReduced = ferrerWork ([], []) (transpMat (snd rRowReduced))
       g           = fst rColReduced
       rRed        = transpMat (snd rColReduced)
   in  (f, rRed, g)

printFerrerDecomposition r =
   let (a,b,c) = ferrerFactorization r
   in  putStr ((stringForOriginalNamedMatrix a) ++
               (stringForOriginalNamedMatrix b) ++
               (stringForOriginalNamedMatrix c))
```

Having this in mind, one will most likely investigate, whether decompositions are also possible if not a linear (strict)ordering but other orderings occur. Michael Winter solved this question positively in his so far unpublished paper *Decomposiing relations into orderings*.

4.3 Line-Covering and Independence

Some relations may be decomposed in such a way, that there is a subset of row entries that is completely unrelated to a subset of column entries. In this context, a relation A may admit vectors x and y (with $\bot\!\!\!\bot \neq x \neq \top\!\!\!\top$ or $\bot\!\!\!\bot \neq y \neq \top\!\!\!\top$ to avoid degeneration), such that $A \mathbin{;} y \subseteq x$ or else $A \subseteq \overline{x} \mathbin{;} \overline{y^\mathsf{T}}$. Given appropriate permutations P of the rows, and Q of the columns, respectively, we then have

$$P \mathbin{;} A \mathbin{;} Q^\mathsf{T} = \begin{pmatrix} * & \bot\!\!\!\bot \\ * & * \end{pmatrix} \qquad P \mathbin{;} x = \begin{pmatrix} \bot\!\!\!\bot \\ \top\!\!\!\top \end{pmatrix} \qquad Q \mathbin{;} y = \begin{pmatrix} \bot\!\!\!\bot \\ \top\!\!\!\top \end{pmatrix}.$$

Given $A;y \subseteq x$, to enlarge the $\amalg$-zone is not so easy a task, which may be seen at the identity relation $\mathbb{I}$: All shapes from $1 \times (n-1)$, $2 \times (n-2)$, $\ldots (n-1) \times 1$ may be chosen. There is no easily acceptable extremality criterion. Therefore, one usually studies this effect with one of the sets negated.

4.3.1 Definition. Let a relation A be given and consider pairs of subsets (s,t) taken from the domain and from the range side, respectively:

i) (s,t) is a **line-covering** $\quad :\Longleftrightarrow \quad A;\overline{t} \subseteq s.$

ii) (s,t) is a **pair of independent sets** $\quad :\Longleftrightarrow \quad A;t \subseteq \overline{s}.$ □

An easy consequence of the definition is the following statement.

4.3.2 Proposition. For a given relation A together with a pair (s,t) we have

$(s,\overline{t})$ line-covering $\iff$ $(\overline{s},t)$ is independent. □

Both concepts allow for enlarging the pair, or reducing it, corresponding to the respective ordering, so as to arrive at an equation. We prove this for a line-covering.

4.3.3 Proposition. i) Let (s,t) be a line-covering of the relation A, i.e., $A;\overline{t} \subseteq s$. Precisely when both, $A;\overline{t} = s$ and $A^\mathsf{T};\overline{s} = t$, are satisfied there will be no smaller (i.e. satisfying both, $x \subseteq s$ and $y \subseteq t$) pair $(x,y) \neq (s,t)$ line-covering A.

ii) Let (s,t) be a pair of independent sets of the relation A, i.e., $A;t \subseteq \overline{s}$. Precisely when both, $A;t = \overline{s}$ and $A^\mathsf{T};s = \overline{t}$, are satisfied there will be no greater (i.e. satisfying both, $x \supseteq s$ and $y \supseteq t$) pair of independent sets $(x,y) \neq (s,t)$ for A.

Proof: i) Assume $A;\overline{t} \subsetneqq s$. Then there will exist a point $p \subseteq s \cap \overline{A;\overline{t}}$ (at least in the case of finite representable relations handled here). Then $x := s \cap \overline{p} \neq s$ and $y := t$ will obviously constitute a strictly smaller line-covering.

If on the other hand side (x,y) is line-covering A with $x \subseteq s$ and $y \subseteq t$ and the two equations are satisfied, then $t = A^\mathsf{T};\overline{s} \subseteq A^\mathsf{T};\overline{x} \subseteq y$. □

Note that we need both of the two equations as $A;y = x$ is not via Schröder's rule equivalent with $A^\mathsf{T};\overline{x} = \overline{y}$.

We provide functions that may — at least for small examples — be used to compute line-coverings and independent sets, respectively. In addition, the inclusion maxi-/minimal and the maxi-/minimal ones by cardinality may be obtained.

```
allInclMinCoverings q =
   let antitoneFctlCovering b x = b *** (negaMat x)
       filterFunction t = t == antitoneFctlCovering (transpMat q)
                                                    (antitoneFctlCovering q t)
       allColVects = allNByM (cols q) 1
   in map (\b -> (head $ transpMat $ antitoneFctlCovering q b,
                  head $ transpMat $ b))
          (filter filterFunction allColVects)

allCardMinCoverings q =
```

```
   let lengthTogether (u,v) = length (filter (== True) (u ++ v))
       minCard = foldl min 100000 (map lengthTogether (allInclMinCoverings q))
   in  filter (\(x,y) -> minCard == lengthTogether (x,y)) (allInclMinCoverings q)

printResMatrixCovering q =
   let pairToSubdivision (u,v) = ([map not u, u], [v, map not v])
       allMinCardCov = map pairToSubdivision (allCardMinCoverings q)
       oneToString (rs, cs) = stringForNamedMatrixLines $ rearrangeMatWithLines q rs cs
   in  putStr ((stringForOriginalNamedMatrix q) ++
               (concat (map oneToString allMinCardCov)))
```

The diversity of reductions shown suggests to look for the following line-covering possibility. For the moment, call rows and columns, respectively, lines. Then together with the $|x|$ by $|y|$ zone of $\mathbf{0}$'s, we are able to cover all entries $\mathbf{1}$ by $|\overline{x}|$ vertical plus $|\overline{y}|$ horizontal lines. It is standard, to try to minimize the number of lines to cover all $\mathbf{1}$'s of the relation.

4.3.4 Definition. Given a relation A, the **term rank** is defined as the minimum number of lines necessary to cover all entries $\mathbf{1}$ in A, i.e.

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

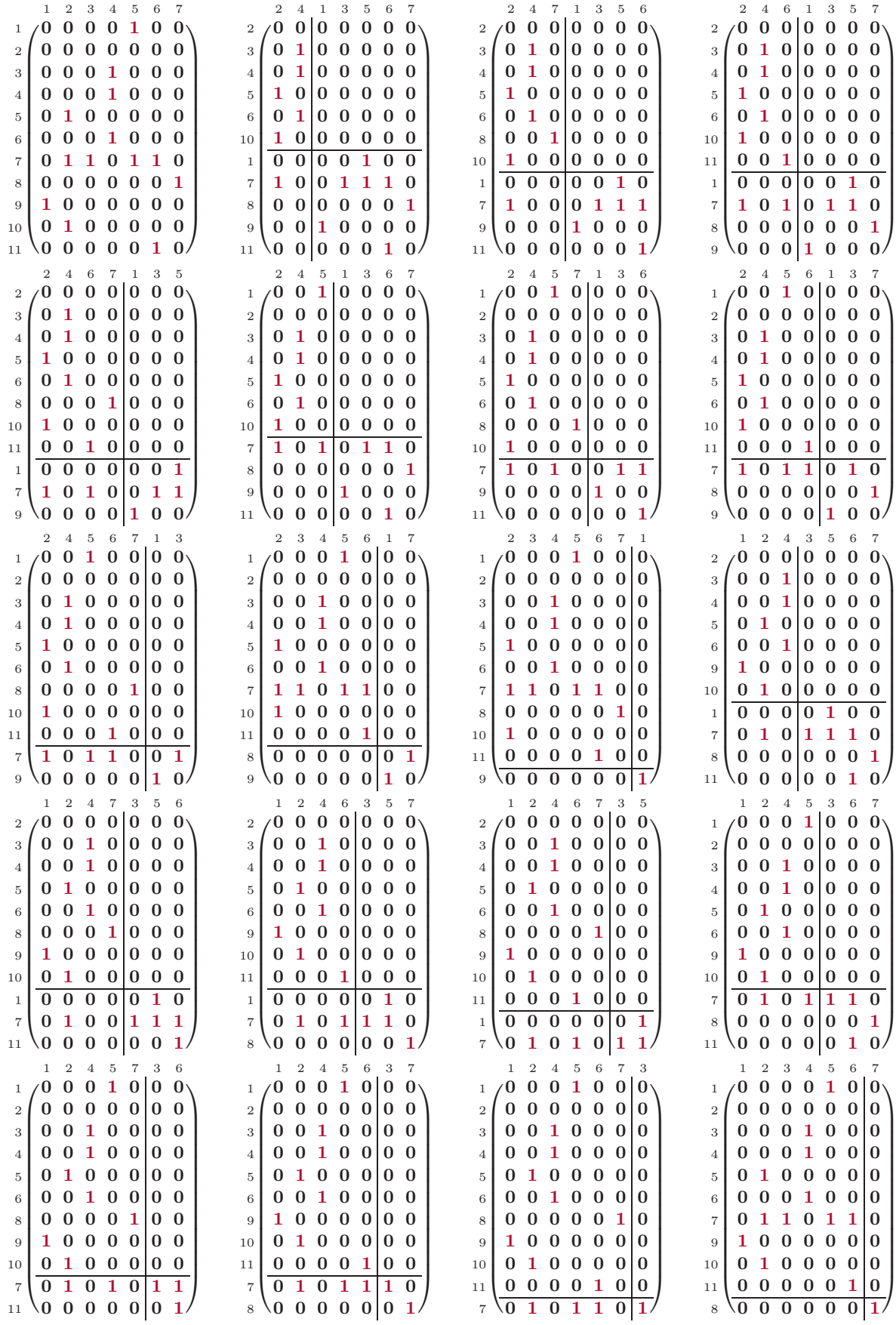
Consider $\begin{pmatrix} A_{11} & \amalg \\ A_{21} & A_{22} \end{pmatrix} \mathbin{;} \begin{pmatrix} \amalg \\ \top \end{pmatrix} = \begin{pmatrix} \amalg \\ \top \end{pmatrix}$. Hoping to arrive at fewer lines than the columns of A_{11} and the rows of A_{22} to cover, one might start a first naive attempt and try to cover with s and t but row i, e.g., omitted. If (s,t) has already been minimal, there will be an entry in row i of A_{22} containing a $\mathbf{1}$. Therefore, A_{22} is a total relation. In the same way, A_{11} turns out to be surjective.

But we may also try to get rid of a set $x \subseteq s$ of rows and accept that a set of columns be added instead. It follows from minimality that regardless of how we choose $x \subseteq s$, there will be at least as many columns necessary to cover what has been left out. This leads to the following famous definition.

4.3.5 Definition. Given a relation A and a set x, we say that x satisfies the **Hall condition**

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

Summarized, if we have a line-covering with $|s| + |t|$ minimal, then A_{11}^{T} as well as A_{22} will satisfy the Hall-condition. We will later learn how to find minimum line-coverings and maximum independent sets without just checking them all exhaustively. Then also a better visualization will become possible; see Page 51. Additional structure will be extracted employing assignment mechanisms. We postpone this, however, until other prerequisites are at hand and concentrate on the following aspect.



A relation of term rank 7 together with all its cardinality maximum reductions

4.3.6 Proposition. Let a finite relation A be given. Then A is either chainable or it admits a pair (s,t) which is nontrivial, i.e., $(s,t) \neq (\bot\!\!\!\bot, \top\!\!\!\top)$ and $(s,t) \neq (\top\!\!\!\top, \bot\!\!\!\bot)$, such that both, s,t as well as $\overline{s},\overline{t}$, constitute at the same time a pair of independent sets and a line-covering.

Proof: Consider the difunctional closure $G := h_{\mathrm{difu}}(A)$. The dichotomy is as to whether $G \neq \top\!\!\!\top$ or $G = \top\!\!\!\top$, in which case A is chainable by definition. Assume the former. We have for arbitrary X always $G ; G^{\mathsf{T}} ; \overline{G ; X} \subseteq \overline{G ; X}$ due to difunctionality. Therefore, every pair $s = G ; X$ and $t = G^{\mathsf{T}} ; \overline{G ; X}$ forms a pair of independent sets of G and all the more of A. However, this holds for $\overline{s} = \overline{G ; X}$ and $\overline{t} = \overline{G^{\mathsf{T}} ; \overline{G ; X}}$ as well, since $G ; \overline{G^{\mathsf{T}} ; \overline{G ; X}} \subseteq G ; X$ using Schröder's rule.

It remains to show that always a nontrivial choice of X may be made, i.e., satisfying both $\neq \bot\!\!\!\bot$ and $\neq \top\!\!\!\top$. If $G = \bot\!\!\!\bot$, we are done as then an arbitrary pair of vectors may be taken. What remains is G difunctional with $G \neq \bot\!\!\!\bot$ and $G \neq \top\!\!\!\top$. Necessary is some case analysis starting from a point outside G, i.e., with $x ; y^{\mathsf{T}} \subseteq \overline{G}$. □

Difunctionality and line-coverings are related basically in the following way.

4.3.7 Proposition. If and only if a relation A admits a pair (x,y) such that (x,y) and $(\overline{x},\overline{y})$ are line-coverings, its difunctional closure will admit these line-coverings.

Proof: Let $H := h_{\mathrm{difu}}(A)$. It is trivial to conclude from H to A as $A \subseteq H$.

From $A ; \overline{y} \subseteq x$ and $A ; \overline{\overline{y}} \subseteq \overline{x}$, or equivalently, $A^{\mathsf{T}} ; x \subseteq \overline{y}$, we derive that $A \subseteq \mathtt{syq}(\overline{x}^{\mathsf{T}}, \overline{y}^{\mathsf{T}})$. As the symmetric quotient is some difunctional relation above A, it is above H, resulting in $H ; \overline{y} \subseteq x$ and $H ; y \subseteq \overline{x}$. □

5 Homogeneous Decompositions

While so far heterogeneous relations have been treated permuting their rows and columns independently, we now specialize to the homogeneous case and apply the permutations simultaneously.

From numerical mathematics we know that square matrices may be reducible, in which case numerical algorithms may perform more efficient. If a reduction of A as $A = \begin{pmatrix} B & 0 \\ C & D \end{pmatrix}$ is known, it becomes easier to solve the linear equation $Ax = b$ which is then $\begin{pmatrix} B & 0 \\ C & D \end{pmatrix} \begin{pmatrix} y \\ z \end{pmatrix} = \begin{pmatrix} c \\ d \end{pmatrix}$. One will solve $By = c$ first and then $Dz = d - Cy$. We shortly investigate irreducible, primitive, and cyclic relations with our techniques.

If a relation is not reducible, it is irreducible or — considering it as a graph — strongly connected. Looking for strongly connected components leads to rather simple decompositions. There are two types of irreducible relations to be investigated, primitive and cyclic ones.

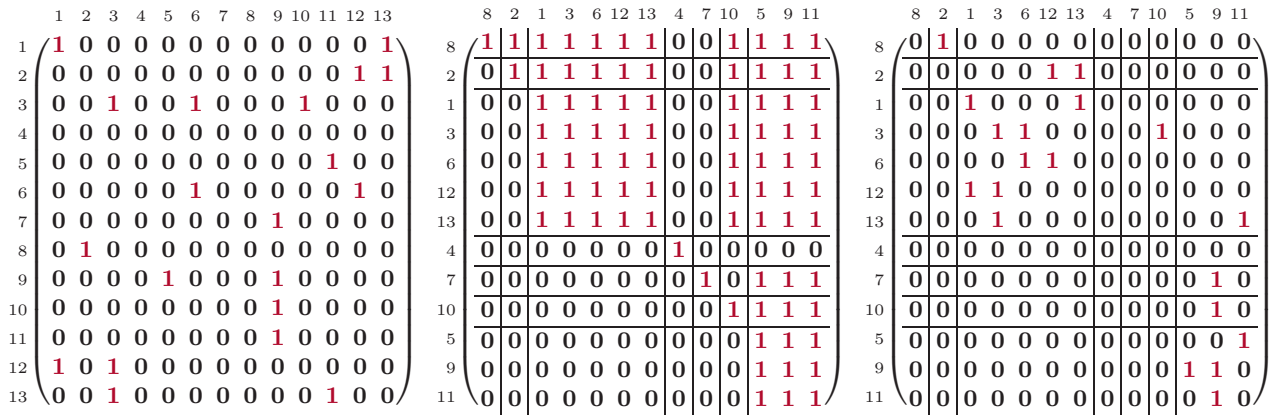
5.1 Decomposing into Strongly Connected Components

Here, the theory behind is more or less well-known. It shall therefore serve as the introduction. Let some relation R be given. Then one may 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.

```
reflTranClosure r = lIdent r ||| (warshall r)
equivGenerated  r = let reflTranClos = reflTranClosure r
                    in  transpMat reflTranClos  &&&  reflTranClos
isStrongConn    r = reflTranClosure r === allMatFor r
```

We now try to find an appropriate decomposition. A good idea is to take the rows of the reflexive transitive closure without their duplicates. When having ordered them lexicographically, we start from the first and eliminate the occurences of `True` in all further rows. This gives the partition to which we rearrange the original matrix. We show the original first, then the preorder generated from it in rearranged form and the rearranged original relation together with the permutation obtained.

```
strongConnCompDecompose r =
   let reflTransClos = reflTranClosure r
       equiClosure   = 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
       grouped r     = rearrangeMatWithLines r rowTypes rowTypes
       closRearr     = stringForNamedMatrixLines $ grouped reflTransClos
       origRearr     = stringForNamedMatrixLines $ grouped r
   in   ((stringForOriginalNamedMatrix r) ++ closRearr ++ origRearr)
```



A relation, the preorder generated from it in rearranged form, and the rearranged original relation

Once a relation is decomposed, one may easily formulate predicate logic theorems capturing what has been achieved. Instead of doing so in full detail already here, we present the following

5.1.1 Proposition (*Decomposing according to strongly connected components*)**.** Any given 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 permuted preorder R^* allows entries $\neq \bot\!\!\bot$. The reflexive-transitive closure of every diagonal block is the universal relation $\top\!\!\top$. □

5.2 Reducible Relations

A relation A on a set X is called **reducible** if there exists a relation $\bot\!\!\bot \neq x \neq \top\!\!\top$, which is *contracted* by A, i.e. $A ; x \subseteq x$. One may say that a relation is reducible, precisely when it contracts (i.e. $A ; x \subseteq x$) some non-trivial (i.e. $\bot\!\!\bot \neq x \neq \top\!\!\top$) relation. Usually one is interested in contracted vectors. Arrows of the graph according to A ending in the subset x will always start in x. It is easy to see that the reducing vectors x, in this case including the trivial ones $x = \bot\!\!\bot$ and $x = \top\!\!\top$, form a lattice.

Using Schröder's rule, a relation A contracts a set x precisely when its tranpose A^T contracts $\overline{x}$ as well: $A ; x \subseteq x \iff A^\mathsf{T} ; \overline{x} \subseteq \overline{x}$. Therefore, a relation is reducible precisely when its tranpose is.

The essence of the reducibility condition is much better visible after determining some permutation P that sends the **1**-entries of x to the end. Applying this *simultaneously* on rows and columns we obtain the shape $P ; A ; P^\mathsf{T} = \begin{pmatrix} B & \bot\!\!\bot \\ C & D \end{pmatrix}$ as well as $P ; x = \begin{pmatrix} \bot\!\!\bot \\ \top\!\!\top \end{pmatrix}$ mentioned in the introduction to this chapter.

The contraction condition can also be expressed as $A \subseteq \overline{x ; x^\mathsf{T}}$. If such a relation x does not exist, A is called **irreducible**. Irreducible means that precisely two vectors are contracted, namely $x = \bot\!\!\bot$ and $x = \top\!\!\top$. As already discussed, a relation is irreducible precisely when its tranpose is. In particular, we have for an irreducible A that $A ; A^* \subseteq A^*$, so that $A^* = \top\!\!\top$, as obviously $A^* \neq \bot\!\!\bot$. Therefore, one can translate this into the language of graph theory:

A irreducible $\iff$ The graph of A is strongly connected.

```
isIrreducible = isStrongConn
```

An irreducible relation A is necessarily total. A certainly contracts $A;\top$ as $A;A;\top \subseteq A;\top$. From irreducibility we obtain that $A;\top = \bot\!\!\!\bot$ or $A;\top = \top$. The former would mean $A = \bot\!\!\!\bot$, so that A would contract every relation x, and, thus, violate irreducibility. Therefore, only the latter is possible, i.e., A is total.

For a reducible relation A and arbitrary k, also A^k is reducible as $A^k;x \subseteq A^{k-1};x \subseteq \ldots \subseteq A;x \subseteq x$. However, the following is an example of an irreducible relation A with A^2 reducible. Therefore, the property of being irreducible is not multiplicative.

$$A = \begin{pmatrix} 0&0&0&1\\ 0&0&0&1\\ 0&0&0&1\\ 1&1&1&0 \end{pmatrix} \quad \begin{pmatrix} 1&1&1&0\\ 1&1&1&0\\ 1&1&1&0\\ 0&0&0&1 \end{pmatrix} = A^2$$

Irreducible relation with reducible square

It is interesting to look for irreducibility of permutations P. We observe $P;x = x$ for $x := \overline{(P^k \cap \mathbb{I});\top}$ and arbitrary k as due to the doublesided mapping properties of P obviously $P;x = P;\overline{(P^k \cap \mathbb{I});\top} = \overline{P;(P^k \cap \mathbb{I});\top} = \overline{(P;P^k \cap P);\top} = \overline{(P^k \cap \mathbb{I});P;\top} = \overline{(P^k \cap \mathbb{I});\top} = x$. For $k = 0$, e.g., this means $x = \bot\!\!\!\bot$ and is rather trivial. Also cases with $P^k \cap \mathbb{I} = \bot\!\!\!\bot$ resulting in $x = \top$ are trivial.

$$\begin{pmatrix} 0&1&0&0&0\\ 0&0&1&0&0\\ 0&0&0&1&0\\ 0&0&0&0&1\\ 1&0&0&0&0 \end{pmatrix} \qquad \left(\begin{array}{cc|ccc} 0&1&0&0&0\\ 1&0&0&0&0\\ \hline 0&0&0&1&0\\ 0&0&0&0&1\\ 0&0&1&0&0 \end{array}\right)$$

Irreducible and reducible permutation

The permutation P is reducible, namely reduced by x, when neither $P^k \cap \mathbb{I} = \bot\!\!\!\bot$ nor $P^k \cap \mathbb{I} = \mathbb{I}$. In recalling permutations, every cycle of P of length c will lead to $P^c \cap \mathbb{I} \neq \bot\!\!\!\bot$. If k is the least common multiple of all cycle lengths occurring in P, obviously $P^k = \mathbb{I}$. If there is just one cycle — as for the cyclic successor relation for $n > 1$ — the permutation P is irreducible. Other permutations with more than one cycle are reducible.

```
cyclSuccessor n = unite [[  replicate (n-1) [False],     ident (n-1)],
                         [  [[True]],                    [replicate (n-1) False]  ]]

--for a bijective mapping simultaneously:
arrangeCyclic m = transpMat m *** (cyclSuccessor (rows m))
--for a bijective mapping permuting columns:
arrangeDiagonal m = transpMat m
```

5.2.1 Proposition. A finite $\quad\Longrightarrow\quad A^n \subseteq (\mathbb{I} \cup A)^{n-1}$ and $(\mathbb{I} \cup A)^{n-1} = A^*$.

Proof: It is trivial that $A^i \subseteq A^*$ for all $i \geq 0$. By the pigeon hole principle always $A^n \subseteq (\mathbb{I} \cup A)^{n-1}$ as otherwise $n+1$ vertices would be needed to delimit a non-selfcrossing path of length n while only n distinct vertices are available. (See Exercise 3.2.6 of [SS89, SS93]) □

5.2.2 Theorem (*See, e.g., [BR96] 1.1.2*)**.** For any boolean $n \times n$-relation A

i) A irreducible $\iff$ $(\mathbb{I} \cup A)^{n-1} = \mathbb{T}$ $\iff$ $A ; (\mathbb{I} \cup A)^{n-1} = \mathbb{T}$

ii) A irreducible $\implies$ There exists an exponent k such that $\mathbb{I} \subseteq A^k$.

Proof: i) We start proving the first equivalence. By definition, A is irreducible, if we have for all vectors $x \neq \mathbb{L}$ that $Ax \subseteq x$ implies $x = \mathbb{T}$. Now, by the preceding proposition $A(\mathbb{I} \cup A)^{n-1} \subseteq (\mathbb{I} \cup A)^{n-1}$ so that indeed $(\mathbb{I} \cup A)^{n-1} = \mathbb{T}$.

For the other direction assume A to be reducible, so that $\mathbb{L} \neq x \neq \mathbb{T}$ exists with $A ; x \subseteq x$. Then also $A^k ; x \subseteq x$ for arbitrary k. This is a contradiction, as it would follow that also $(\mathbb{I} \cup A)^{n-1} ; x \subseteq x$ resulting in $\mathbb{T} ; x \subseteq x$ and, in the case of boolean matrices, i.e., with Tarski's rule satisfied, in $x = \mathbb{T}$, a contradiction.

Now we prove the second equivalence. Using Prop. 5.2.1, $(\mathbb{I} \cup A)^{n-1} \supseteq A ; (\mathbb{I} \cup A)^{n-1}$, so that also $A ; (\mathbb{I} \cup A)^{n-1} \supseteq A ; A ; (\mathbb{I} \cup A)^{n-1}$. Since A is irreducible, this leads to $A ; (\mathbb{I} \cup A)^{n-1}$ being equal to $\mathbb{L}$ or $\mathbb{T}$, from which only the latter is possible.

ii) Consider the irreducible $n \times n$-relation A and its powers. According to (i), there exists for every row number j a least power $1 \leq p_j \leq n$ with position $(j, j) \in A^{p_j}$. For the least common multiple p of all these p_j we have $\mathbb{I} \subseteq A^p$, and p is the smallest positive number with this property. □

$$\left(\begin{array}{cccccccccccc}
0&1&1&0&0&0&0&0&0&0&0&0\\
0&0&0&0&1&1&0&1&0&1&0&0\\
0&0&0&0&0&1&0&0&0&0&0&0\\
0&0&0&0&0&0&1&0&0&0&1&0\\
0&1&0&0&1&1&0&0&0&0&0&0\\
1&0&0&0&0&0&0&0&0&0&0&0\\
1&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&1&0&0&0&0&0&0&1\\
0&0&0&1&0&1&0&0&1&0&0&1\\
0&0&0&1&0&0&1&0&1&0&0&0\\
0&0&0&0&0&0&0&0&0&1&0&0\\
0&1&0&0&0&0&0&0&0&0&0&0\\
0&0&0&1&0&0&0&0&1&1&0&0
\end{array}\right)
\left(\begin{array}{cccccccccccc}
0&0&0&0&1&1&0&1&0&1&0&0\\
1&1&0&1&1&1&1&0&1&0&0&1\\
1&0&0&0&0&0&0&0&0&0&0&0\\
1&0&0&1&0&0&0&0&1&1&0&0\\
1&1&0&0&1&1&0&1&0&1&0&0\\
0&1&1&0&0&0&0&0&0&0&0&0\\
0&1&1&0&0&0&0&0&0&0&0&0\\
0&1&0&0&1&1&0&0&0&0&0&0\\
1&1&0&1&0&1&1&0&1&0&1&1\\
1&0&0&1&0&1&1&0&1&0&1&1\\
0&0&0&1&0&0&1&0&1&0&0&0\\
0&0&0&0&1&1&0&1&0&1&0&0\\
0&0&0&1&0&1&1&0&1&0&1&1
\end{array}\right)
\left(\begin{array}{cccccccccccc}
1&1&0&1&1&1&1&0&1&0&0&1\\
1&1&1&1&1&1&1&1&1&1&1&1\\
0&1&1&0&0&0&0&0&0&0&0&0\\
0&1&1&1&0&1&1&0&1&0&1&1\\
1&1&1&1&1&1&1&1&1&1&0&1\\
0&0&0&0&1&1&0&1&0&1&0&0\\
0&0&0&0&1&1&0&1&0&1&0&0\\
1&1&0&0&1&1&0&1&0&1&0&0\\
1&1&1&1&1&1&1&1&1&1&1&1\\
1&1&1&1&0&1&1&0&1&1&1&1\\
1&0&0&1&0&1&1&0&1&0&1&1\\
1&1&0&1&1&1&1&0&1&0&0&1\\
1&1&0&1&0&1&1&0&1&1&1&1
\end{array}\right)$$

$$\left(\begin{array}{cccccccccccc}
1&1&1&1&1&1&1&1&1&1&1&1\\
1&1&1&1&1&1&1&1&1&1&1&1\\
0&0&0&0&1&1&0&1&0&1&0&0\\
1&1&0&1&1&1&1&1&1&1&1&1\\
1&1&1&1&1&1&1&1&1&1&1&1\\
1&1&0&1&1&1&1&0&1&0&0&1\\
1&1&0&1&1&1&1&0&1&0&0&1\\
1&1&1&1&1&1&1&1&1&1&0&1\\
1&1&1&1&1&1&1&1&1&1&1&1\\
1&1&1&1&1&1&1&1&1&1&1&1\\
1&1&1&1&0&1&1&0&1&1&1&1\\
1&1&1&1&1&1&1&1&1&1&1&1\\
1&1&1&1&1&1&1&1&1&1&1&1
\end{array}\right)
\left(\begin{array}{cccccccccccc}
1&1&1&1&1&1&1&1&1&1&1&1\\
1&1&1&1&1&1&1&1&1&1&1&1\\
1&1&0&1&1&1&1&0&1&0&0&1\\
1&1&1&1&1&1&1&1&1&1&1&1\\
1&1&1&1&1&1&1&1&1&1&1&1\\
1&1&1&1&1&1&1&1&1&1&1&1\\
1&1&1&1&1&1&1&1&1&1&1&1\\
1&1&1&1&1&1&1&1&1&1&1&1\\
1&1&1&1&1&1&1&1&1&1&1&1\\
1&1&1&1&1&1&1&1&1&1&1&1\\
1&1&1&1&1&1&1&1&1&1&1&1\\
1&1&1&1&1&1&1&1&1&1&1&1\\
1&1&1&1&1&1&1&1&1&1&1&1
\end{array}\right)
\left(\begin{array}{cccccccccccc}
1&1&1&1&1&1&1&1&1&1&1&1\\
1&1&1&1&1&1&1&1&1&1&1&1\\
1&1&1&1&1&1&1&1&1&1&1&1\\
1&1&1&1&1&1&1&1&1&1&1&1\\
1&1&1&1&1&1&1&1&1&1&1&1\\
1&1&1&1&1&1&1&1&1&1&1&1\\
1&1&1&1&1&1&1&1&1&1&1&1\\
1&1&1&1&1&1&1&1&1&1&1&1\\
1&1&1&1&1&1&1&1&1&1&1&1\\
1&1&1&1&1&1&1&1&1&1&1&1\\
1&1&1&1&1&1&1&1&1&1&1&1\\
1&1&1&1&1&1&1&1&1&1&1&1\\
1&1&1&1&1&1&1&1&1&1&1&1
\end{array}\right)$$

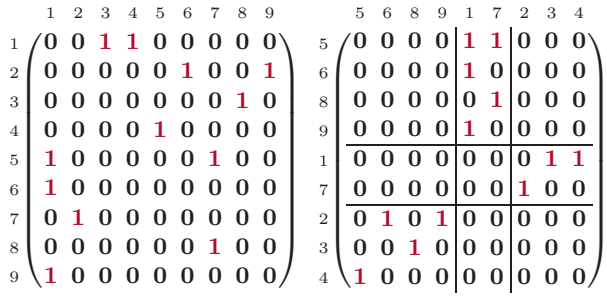
Powers $R^1, \ldots, R^6$ of a primitive relation R

We now classify irreducible relations a bit further and consider two important classes.

5.2.3 Definition (*See, e.g., [Var62] 2.5*)**.** An irreducible relation A is called **primitive** if there exists some integer $k \geq 1$ such that $A^k = \mathbb{T}$. If this is not the case, the irreducible relation may be called **cyclic of order** k, indicating that the (infinitely many) powers $A, A^2, A^3 \ldots$ reproduce cyclically and k is the greatest common divisor of all the occurring periods. □

We observe first powers of primitive relations.

```
primResult iR = let powers = [ powerOf iR i | i <- [0..] ]
                    newPowers  = nub (take (cols iR) powers)
                    printPower  = map druckTeXMatH newPowers
                    stringPower = foldl (\ x y -> x ++ "$\n\n\n$" ++ y) "" printPower
                in  putStr stringPower
```



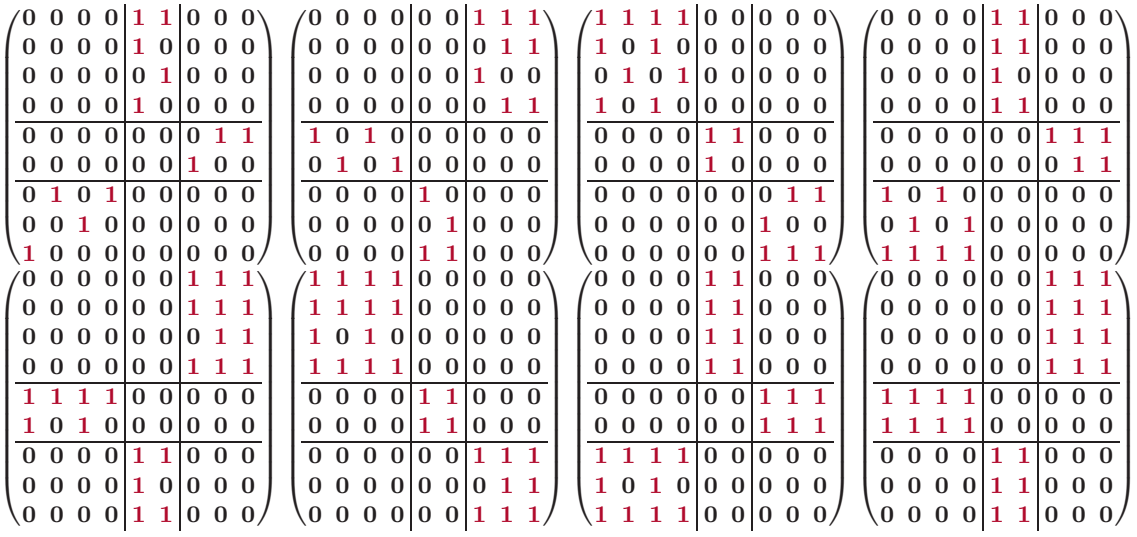
A cyclic relation and its rearranged version

5.2.4 Proposition (*See, e.g., [BR96] 1.8.2*)**.** A relation R is primitive precisely when its powers R^k are irreducible for all $k \geq 1$.

Proof: "$\Longrightarrow$": Assume $R^k ; x \subseteq x$ with $x \neq \perp\!\!\!\perp$ and $x \neq \top\!\!\!\top$ for some $k \geq 1$. Then we have also $R^{nk} ; x \subseteq R^{(n-1)k} ; x \ldots \subseteq R^k ; x \subseteq x$ for all $n \geq 1$. This contradicts primitivity of R as from some index on all powers of a primitive R will be $\top\!\!\!\top$.

"$\Longleftarrow$": Assume R were not primitive, i.e. $R^k \neq \top\!\!\!\top$ for all k. It is impossible for any R^k to have a column $\perp\!\!\!\perp$ as this would directly show reducibility of R^k. It follows from finiteness that there will exist identical powers $R^l = R^k \neq \top\!\!\!\top$ with $l > k$, e.g. This results in $R^{l-k} ; R^k = R^k$. Power R^{l-k}, therefore, contracts all columns of R^k — and at least one column which is unequal $\top\!\!\!\top$. □

Now we observe how powers of a cyclic relation behave.



Rearranged powers $R^1, \ldots R^8$ of a cyclic relation R

The behaviour of irreducible relations just presented may be summarized as follows. In both cases, rectangular blocks are gradually filled. If this is completed, one will find a difunctional matrix. In the primitive case, it is the universal matrix, while for a cyclic relation it is a permutation of these blocks with just one long period. We will only later be able to look for algebraic reasons for this behaviour.

5.3 Difunctionality and Irreducibility

We now specialize our investigations concerning difunctional relations to homogeneous and, later, irreducible ones.

5.3.5 Proposition. Let an arbitrary finite homogeneous relation R be given. Then in addition to the constructs Ξ, Ξ' of Prop. 4.1.4 also $\Theta := (\Xi \cup \Xi')^*$, $G := \Theta ; R$, and $G' := R ; \Theta$ may be formed.

i) Θ is an equivalence.

ii) $G ; G^\mathsf{T} \subseteq \Theta$ and $G'^\mathsf{T} ; G' \subseteq \Theta$

iii) G as well as G' are difunctional.

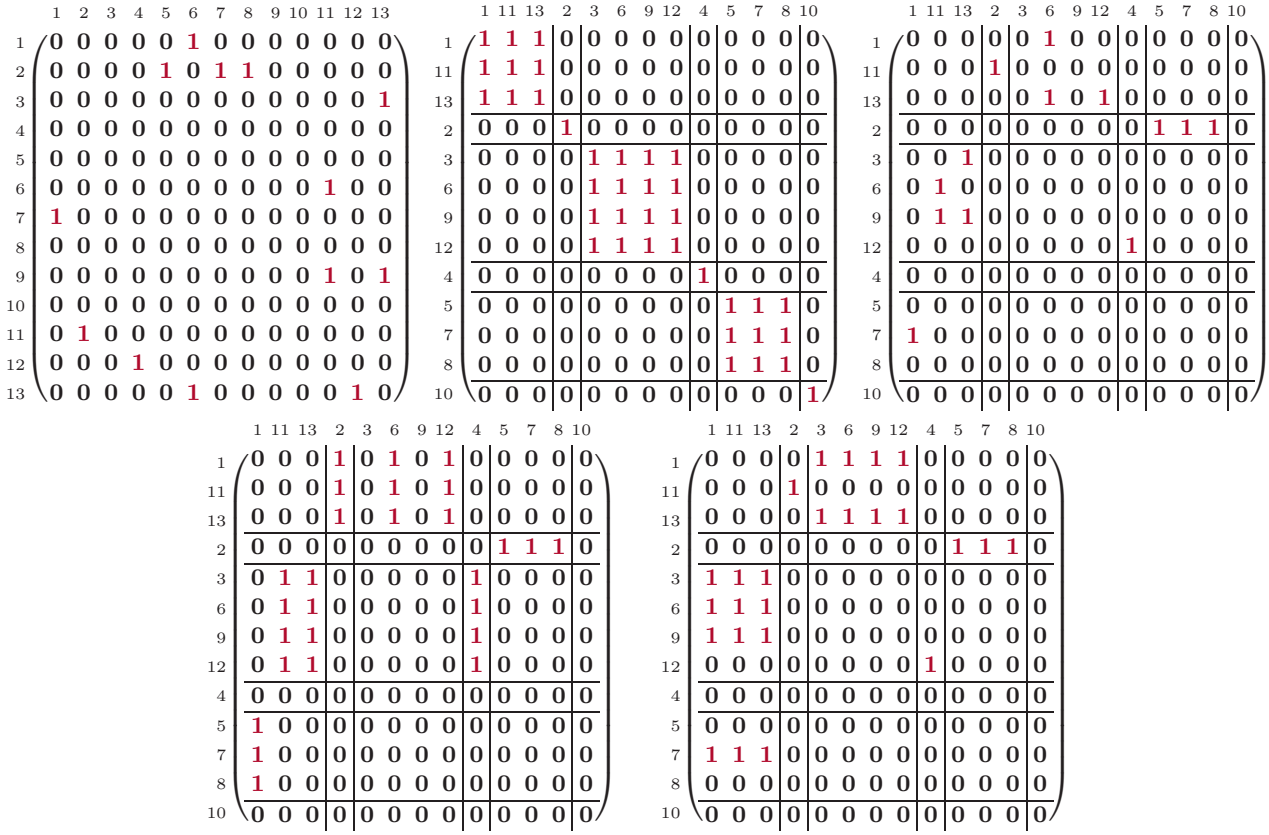
Proof: (i) is trivial. From the other statements, we prove the G-variants.
(ii) $G ; G^\mathsf{T} = \Theta ; R ; (\Theta ; R)^\mathsf{T} = \Theta ; R ; R^\mathsf{T} ; \Theta \subseteq \Theta ; \Xi ; \Theta \subseteq \Theta ; \Theta ; \Theta = \Theta$
(iii) $G ; G^\mathsf{T} ; G \subseteq \Theta ; G = \Theta ; \Theta ; R = \Theta ; R = G$, using (ii). □

It need not be that $G^\mathsf{T} ; G \subseteq \Theta$; see the example $R = \begin{pmatrix} \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{1} & \mathbf{1} \\ \mathbf{1} & \mathbf{0} & \mathbf{0} \end{pmatrix}$ with $G = \begin{pmatrix} \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{1} & \mathbf{1} & \mathbf{1} \\ \mathbf{1} & \mathbf{1} & \mathbf{1} \end{pmatrix}$. Nor need the pair (Θ, Θ) be an R-congruence as this example shows, where also $G \neq G'$.

```
difuSymmClosure r =
   let transpR = transpMat r
       xi      = reflTranClosure $ r *** transpR
       xi'     = reflTranClosure $ transpR *** r
   in  reflTranClosure $ xi ||| xi'
```

With the following function we visualize what happens.

```
printResDifuSymm r =
   let theta  = difuSymmClosure r
       ggggggg = theta *** r
       gPrime = r *** theta
       rsH    = nub (transpMat theta)
       rs     = if and (map or (transpMat rsH))
                then  reverse $ sort $ rsH
                else (reverse $ sort $ rsH) ++ [map not (map or (transpMat rsH))]
       origRearr        = stringForNamedMatrixLines $ rearrangeMatWithLines r      rs rs
       symDifuClosRearr = stringForNamedMatrixLines $ rearrangeMatWithLines theta  rs rs
       gggggggRearr = stringForNamedMatrixLines $ rearrangeMatWithLines ggggggg rs rs
       gPrimeRearr = stringForNamedMatrixLines $ rearrangeMatWithLines gPrime rs rs
   in  ((stringForOriginalNamedMatrix r) ++ symDifuClosRearr ++
              origRearr ++ ggggggRearr ++ gPrimeRearr)
```



A relation, its symmetrized difunctional closure, rearranged according to it, and G, G'

One easily observes, that the relation is not yet block-wisely injective, nor need it be block-wisely univalent. The blocks of G, G' are not yet completely filled, but only filled row- or columnwise, respectively. So by applying the permutations simultaneously, we have lost some of the properties the relations enjoyed when permuting independently in the heterogeneous case. In the next theorem, we define a bigger congruence where we get back what has just been lost.

5.3.6 Proposition. Let a finite and homogeneous relation R be given, and consider the constructs Ξ, Ξ' of Prop. 4.1.4 and $\Theta := (\Xi \cup \Xi')^*$ as in Prop. 5.3.5. Define Ω as the stationary value of the iteration

$$X \mapsto \tau(X) := (X \cup R ; X ; R^\mathsf{T} \cup R^\mathsf{T} ; X ; R)^*$$

started with $X_0 := \mathbb{I}$.

i) Ω is an equivalence containing Ξ, Ξ', and Θ.

ii) "Considered modulo Ω", the relation R is

univalent $R^\mathsf{T} ; \Omega ; R \subseteq \Omega$, and

injective $R ; \Omega ; R^\mathsf{T} \subseteq \Omega$.

iii) $H := \Omega ; R ; \Omega$ is difunctional and commutes with Ω, i.e., $\Omega ; H = H ; \Omega$, so that the pair (Ω, Ω) constitutes an H-congruence.

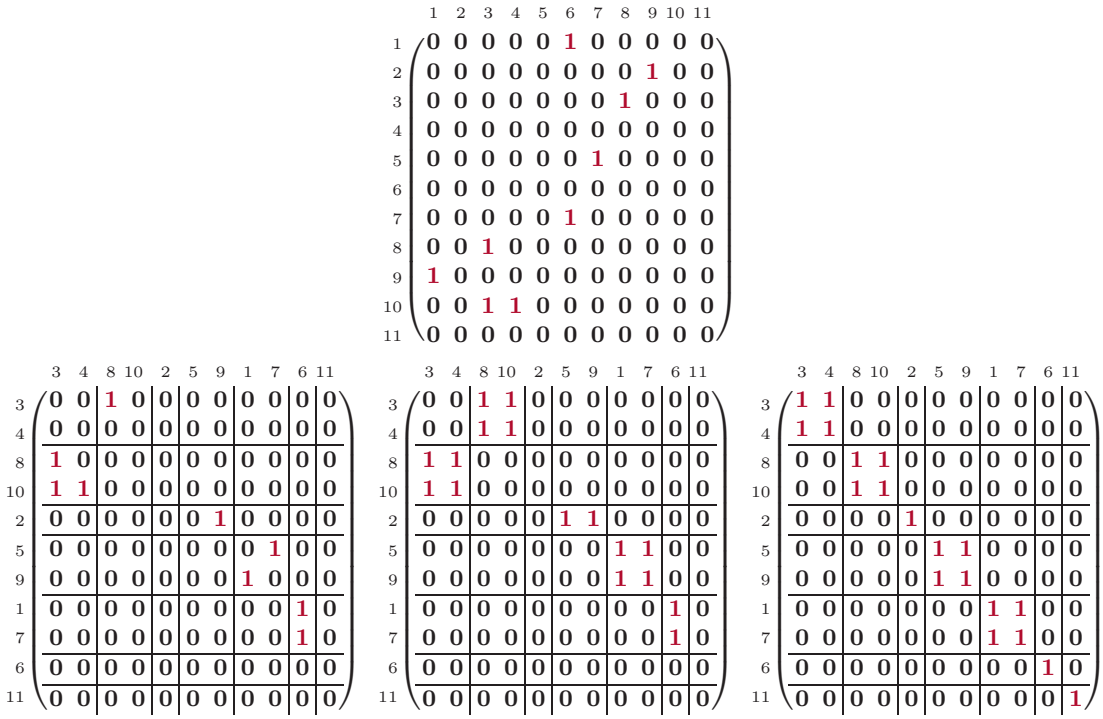
Proof: i) The isotone iteration $X \mapsto \tau(X)$ will end after a finite number of steps with a relation Ω satisfying $\mathbb{I} \subseteq \Omega = \tau(\Omega) = (\Omega \cup R ; \Omega ; R^\mathsf{T} \cup R^\mathsf{T} ; \Omega ; R)^*$. Thus, Ω is reflexive, symmetric, and transitive

by construction, i.e., it is an equivalence. This equivalence certainly contains $R;R^\mathsf{T}$, and therefore, Ξ. Also $\Xi' \subseteq \Omega$ in an analogous way, so that it also contains Θ.
ii) is trivial.
iii) $H;H^\mathsf{T};H = \Omega;R;\Omega;(\Omega;R;\Omega)^\mathsf{T};\Omega;R;\Omega = \Omega;R;\Omega;R^\mathsf{T};\Omega;R;\Omega \subseteq \Omega;R;\Omega;\Omega;\Omega = \Omega;R;\Omega = H$
$\Omega;H = \Omega;\Omega;R;\Omega = \Omega;R;\Omega = \Omega;R;\Omega;\Omega = H;\Omega$ □

Another characterization is
$\Omega = \mathtt{inf}\{Q \mid Q \text{ equivalence}, Q;R \subseteq R;Q, R^\mathsf{T};R \subseteq Q, R;R^\mathsf{T} \subseteq Q,\}$. So (Ω,Ω) is the smallest R-congruence above (Ξ,Ξ').

The following function allows to compute this closure.

```
difuClosLeftRightIterated r =
   let transpR = transpMat r
       tauR x  = reflTranClosure $ (x |||
                    (transpR *** (x *** r))) ||| (r *** (x *** transpR))
       untilT f x = let fx = f x
                    in  if (x == fx) then x
                                     else untilT f fx
   in  untilT tauR (ident (rows r))
```



A relation with its rearranged and block-filled form according to Ω

One will observe the block-successor form of Page 12 with a 2-cycle first and then a terminating strand of 4.

```
printLeftRightIterated r =
   let omega = difuClosLeftRightIterated r
       hhhhh = omega *** (r *** omega)
       rs    = nub omega
       rno   = rows rs
       (aaa,_,_,_,_) = rearrangeMatWithLines hhhhh rs rs
```

```
    aaanub = nub aaa
    aaanubnub = transpMat (nub (transpMat aaanub)) -- this is now injective and univalent
    ran = rows aaanubnub
    can = cols aaanubnub
    fullagain = unite [[aaanubnub,                nulMatNM ran (rno - ran)      ],
                       [nulMatNM (rno - ran) can, nullMatFor $ ident (rno - ran)]]
    succForm = permToSuccMatrix fullagain          -- of the groups
    rsCyc = succForm *** rs
    omegaRearr     = stringForNamedMatrixLines $ rearrangeMatWithLines omega rsCyc rsCyc
    origRearr      = stringForNamedMatrixLines $ rearrangeMatWithLines r     rsCyc rsCyc
    omegaClosRearr = stringForNamedMatrixLines $ rearrangeMatWithLines hhhhh rsCyc rsCyc
in  ((stringForOriginalNamedMatrix r) ++ origRearr ++ omegaClosRearr ++ omegaRearr)
```

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
1	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	1	0	0	0
2	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0
3	0	0	0	0	0	1	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	1	1	0	1	0	0	0	0	0	0	0	0	1	0
6	0	0	0	0	0	0	0	0	0	1	0	0	0	0	1	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0
8	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	1	0	0	0
9	0	0	1	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
11	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0
12	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
13	0	0	0	1	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
14	0	0	0	0	0	1	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0
15	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
16	0	0	0	0	0	0	0	0	0	1	0	0	0	0	1	0	0	0	0	0	0
17	1	0	0	0	0	0	0	1	1	0	1	0	0	0	0	0	0	0	0	1	0
18	0	0	0	1	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
19	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
20	0	0	1	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0
21	1	0	0	0	0	0	0	1	1	0	1	0	0	0	0	0	0	0	0	0	0

	1	8	9	11	20	2	3	13	14	18	4	6	7	12	16	10	15	19	5	17	21
1	0	0	0	0	0	0	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0
11	0	0	0	0	0	1	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0
20	0	0	0	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	1	0	1	0	1	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0	0	0	0	0	0	0
13	0	0	0	0	0	0	0	0	0	0	1	1	0	1	0	0	0	0	0	0	0
14	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0	0	0	0	0	0	0
18	0	0	0	0	0	0	0	0	0	0	1	1	0	1	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0
7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0
12	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0
10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0
19	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
5	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
17	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
21	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

An irreducible and cyclic relation in original and rearranged form

Very often Ω will be much too big an equivalence, close to $\top$, to be interesting. There are special cases, however, where we encounter the well-known Moore-Penrose configuration again.

5.3.7 Proposition. Assume the settings of Prop. 5.3.6, and assume that R in addition be irreducible. Then the following hold:

i) H is irreducible.

ii) H is total and surjective making it a 1:1-correspondence of the classes according to Ω.

iii) $H^\intercal$ acts as an "inverse" of H in as far as $H^\intercal ; H = \Omega$, as well as $H^\intercal ; H^2 = H$, etc.

iv) There exists a power k such that $R^k = \Omega$ and $R^{k+1} = H$.

Proof: (i) Assumed $H ; x \subseteq x$ to hold, then all the more $R ; x \subseteq x$. If it were the case that $\bot \neq x \neq \top$, we would obtain that R were reducible, i.e., a contradiction.
(ii) As proved on page 34 directly after the definition of irreducibility, an irreducible relation R is total. This holds for $\Omega ; R ; \Omega$ as well. Surjectivity is shown analogously.

(iii) We now easily deduce that $H^\intercal ; H = \Omega$, since with surjectivity and definition of Ω
$H^\intercal ; H = \Omega ; R^\intercal ; \Omega ; \Omega ; R ; \Omega = \Omega ; R^\intercal ; \Omega ; R ; \Omega = \Omega ; \Omega ; \Omega = \Omega$.

Then also by induction
$H^\intercal ; H^{k+1} = H^\intercal ; H ; H^k = \Omega ; H^k = H^k$.

(iv) We may prove a property we had already on page 34 for permutations proper, i.e., with all block-widths 1: The construct $x := \overline{(H^k \cap \Omega) ; \top}$ satisfies $H ; x \subseteq x$ for all powers k.
$H ; x = H ; \overline{(H^k \cap \Omega) ; \top} = \overline{H ; (H^k \cap \Omega) ; \top} = \overline{(H^{k+1} \cap H) ; \top} = \overline{(H^k \cap \Omega) ; H ; \top} = \overline{(H^k \cap \Omega) ; \top} = x$

To this end, we convince ourselves that $\overline{H^\intercal ; X} = H^\intercal ; \overline{X}$ proving the second equality, and $H ; (H^k \cap \Omega) = H^{k+1} \cap H = (H^k \cap \Omega) ; H$ proving the others.

From totality of $H^\intercal$ we have $\top = H^\intercal ; \top = H^\intercal ; X \cup H^\intercal ; \overline{X}$ so that always $\overline{H^\intercal ; X} \subseteq H^\intercal ; \overline{X}$. The opposite inclusion is satisfied for arbitrary X with $\Omega ; X = X$, since $H ; H^\intercal ; X \subseteq \Omega ; X = X$.

Furthermore, we have
$H ; (H^k \cap \Omega) \subseteq H ; H^k \cap H \subseteq (H \cap H ; H^{k\intercal}) ; (H^k \cap H^\intercal ; H) \subseteq H ; (H^k \cap H^\intercal ; H) = H ; (H^k \cap \Omega)$
giving equality everywhere in between.

We have, after all, that $R ; x \subseteq H ; x = x$, regardless of how we choose k. However, R is irreducible, which means that no x unequal to $\bot, \top$ is allowed to occur. This restricts H^k to be either Ω or to be disjoint therefrom. □

6 Galois-Decompositions

There is a well-developed theory of standard iterations for boolean matrices in order to solve a diversity of application problems, such as matching, line-covering, assignment, games, etc. We will present a general framework for executing these iterations.

When studying, e.g., reducibility, we found that there may exist many reductions. In order to not take an arbitrary one, we should know additional properties we demand for them. One such property is provided by the termination condition. Another one arises in combination with the well-known matching/assignment situation.

6.1 Galois-Iterations in General

In all of these cases, we need two antitone mappings, which we call $\sigma : \mathcal{P}(V) \to \mathcal{P}(W)$ and $\pi : \mathcal{P}(W) \to \mathcal{P}(V)$, according to [SS89, SS93]. These mappings are usually determined by an obviously antitone relational construct based on some relations, e.g., $w \mapsto \pi(w) := \overline{B;w}$ based on the relation $B : V \leftrightarrow W$. Many other pairs of such antitone mappings are conceivable.

Nested iterations will then start with the empty subset of V on the left and the full subset of W on the right[1]. 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 $\mathbb{\bot} \subseteq \pi(\mathbb{T})$ and $\sigma(\mathbb{\bot}) \subseteq \mathbb{T}$ which are perpetuated by the antitone mappings to $\mathbb{\bot} \subseteq \pi(\mathbb{T}) \subseteq \pi(\sigma(\mathbb{\bot})) \subseteq \ldots$ and $\ldots \subseteq \sigma(\pi(\mathbb{T})) \subseteq \sigma(\mathbb{\bot}) \subseteq \mathbb{T}$. In the finite case, these two sequences will eventually become stationary. The effect of the iteration is that the least fixedpoint a of $v \mapsto \pi(\sigma(v))$ on the side started with the empty set is related to the greatest fixedpoint b of $w \mapsto \sigma(\pi(w))$ on the side started from the full set. The final situation obtained will be characterised by $a = \pi(b)$ and $\sigma(a) = b$.

Of course, there exist fixedpoints of $v \mapsto \pi(\sigma(v))$ and $w \mapsto \sigma(\pi(w))$, as these are isotone mappings in a complete powerset lattice. These fixedpoints are reached since the iteration starts with trivial containments $\mathbb{\bot} \subseteq x$ and $y \subseteq \mathbb{T}$ for arbitrary fixedpoints $x = \pi(\sigma(x))$ and $y = \sigma(\pi(y))$, which are again perpetuated by the iteration to $\mathbb{\bot} \subseteq \pi(\mathbb{T}) \subseteq \pi(y)$ and $\sigma(x) \subseteq \sigma(\mathbb{\bot}) \subseteq \mathbb{T}$, and furthermore to $\mathbb{\bot} \subseteq \pi(\mathbb{T}) \subseteq \pi(\sigma(\mathbb{\bot})) \subseteq \pi(\sigma(x)) = x$ and $y = \sigma(\pi(y)) \subseteq \sigma(\pi(\mathbb{T})) \subseteq \sigma(\mathbb{\bot}) \subseteq \mathbb{T}$, etc.

We formulate the basic iteration along the well-known `until`-construct of Haskell with `lr` for σ and `rl` for π.

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

For detailed proofs concerning this iteration see A.3.11 of [SS89, SS93]. The start may be determined from the row and column number of the given basic relations inserting `True, False` as appropriate.

[1] Of course, the iteration may also be started the other way round, i.e., with the full subset of V on the left and with the empty subset of W on the right. Then corresponding results are obtained for B^{T}.

```
startVector tf b = map (\x -> [tf]) b
```

We will in the sections to come use this general scheme in several application fields.

6.2 Termination

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

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

is called the **initial part** $J(R)$ of R. We will now determine the initial part of a relation along the lines of 6.3.4 of [SS89, SS93]. A relation is *progressively finite* if $J(R) = \top$. A slightly different property is being *progressively bounded*, $\mathtt{sup}_{h\geq 0}\, \overline{B^h;\top} = \top$. A difference between the two exists only for non-finite relations; it may, thus, be neglected here.

Two antitone functionals $v \mapsto \sigma(v) := \overline{v}$ and $w \mapsto \pi(w) := R;\overline{w}$ are given as follows:

```
antitoneFctlCorr1 r v = negaMat v          -- independent of r!
antitoneFctlCorr2 r w = r *** (negaMat w)
```

Applying the general scheme from the last section, we obtain the initial part as

```
initialPart r = untilGS (antitoneFctlCorr2 r)
                        (antitoneFctlCorr1 r)
                        (startVector False r, startVector True r)
```

The algorithm applied to the relation R will result in the 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.)

$$R = \begin{pmatrix} 0&0&0&0&0&0&0&1&0\\ 0&0&0&0&0&0&0&0&1\\ 0&0&0&0&0&0&1&0&0\\ 0&1&0&0&0&0&0&0&0\\ 0&1&0&1&0&0&0&0&1\\ 1&0&0&0&0&0&1&0&0\\ 0&0&0&0&0&0&0&0&1\\ 1&0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&0&0 \end{pmatrix} \qquad a = \begin{pmatrix} 1\\0\\0\\0\\0\\1\\0\\1\\0 \end{pmatrix} \qquad b = \begin{pmatrix} 0\\1\\1\\1\\1\\0\\1\\0\\1 \end{pmatrix}$$

A relation together with the results of the termination iteration

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 offers the possibility of rearranging as follows:

6.2.1 Proposition (*Rearranging relations with respect to termination*)**.** 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:

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

□

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

$$a = \begin{pmatrix} \text{⫫} \\ \top \end{pmatrix} = \begin{pmatrix} \text{progressively bounded} & \text{⫫} \\ * & \text{total} \end{pmatrix};\overline{\begin{pmatrix} \top \\ \text{⫫} \end{pmatrix}} \qquad b = \begin{pmatrix} \top \\ \text{⫫} \end{pmatrix} = \overline{\begin{pmatrix} \text{⫫} \\ \top \end{pmatrix}}$$

In as far as $R;a = R;\overline{b} = a$, we got that R is reducible. Among the many possible ones we have distinguished a reduction with very specific properties. We look for yet another example of rearranging and subdividing a relation according to termination properties.

```
printResTermination m =
   let terminationResult = initialPart m
       aa = map head (fst terminationResult)
       bb = map head (snd terminationResult)
       rowCols = [aa, bb]
       origRearr = stringForNamedMatrixLines $ rearrangeMatWithLines m rowCols rowCols
   in  stringForOriginalNamedMatrix m ++ origRearr
```

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
1	0	0	1	0	0	0	0	1	0	0	0	0	0	1	0	0	0
2	0	0	0	0	0	1	0	1	0	0	0	0	0	0	1	1	1
3	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
4	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0
5	1	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
8	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
9	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
11	0	0	1	0	0	0	0	0	1	0	0	1	0	1	0	0	0
12	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
13	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
14	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0
15	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0
16	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
17	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	3	7	12	17	1	2	4	5	6	8	9	10	11	13	14	15	16
3	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
12	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
17	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0	0	1	0	0	0	0	1	0	0
2	0	0	0	1	0	0	0	0	1	1	0	0	0	0	0	1	1
4	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0
5	0	1	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0
8	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1
9	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
11	1	0	1	0	0	0	0	0	0	0	1	0	0	0	1	0	0
13	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
14	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
15	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
16	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0

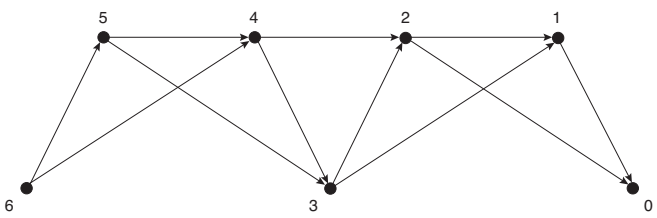
A relation, original and rearranged according to its initial part

The termination-oriented decomposition may prove useful in the following case: Assume a preference relation to be given where it is not clear from the beginning that this preference is circuit-free. There is a tendency of ranking all those equal who belong to a circuit. The initial part collects all items that are not related to a circuit at all, so that they are properly ranked by the given relation.

6.3 Games

For a second application, we look at solutions of relational games. Let an arbitrary homogeneous relation $B : V \leftrightarrow V$ be given. Two players are supposed to make moves alternatingly according to B in choosing a consecutive arrow to follow. The player who has no further move, i.e., who is about to move and finds an empty row in the relation B, or a terminal vertex in the graph, has lost.

Such a game is easily visualized taking a relation B represented by a graph, on which players have to determine a path in an alternating way. We study it for the Nim-type game starting with 6 matches from which we are allowed to take 1 or 2.



	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	1	0	0	0	0	0	0
2	1	1	0	0	0	0	0
3	0	1	1	0	0	0	0
4	0	0	1	1	0	0	0
5	0	0	0	1	1	0	0
6	0	0	0	0	1	1	0

A NIM-type game in graph- and in matrix-form

The antitone functionals based on this relation are formed in a manner quite similar to termination.

```
antitonFctlGame b x = negaMat (b *** x)
```

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

```
gameSolution b = untilGS (antitonFctlGame b) (antitonFctlGame b)
                         (startVector False b, startVector True b)
```

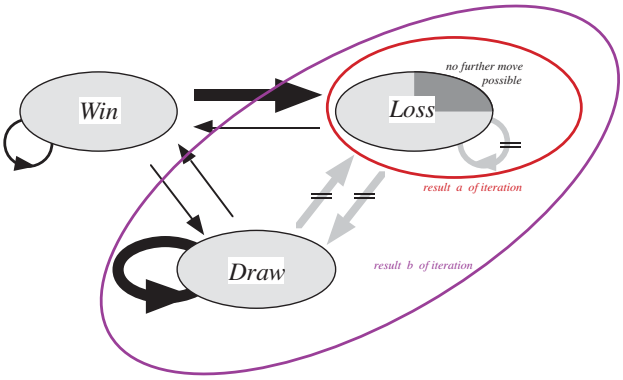
There is one further point to mention concerning the result. This time, we have a homogeneous relation, and we easily observe, that the two sequences from page 42 reduce using monotony to just one

$$\bot\!\!\bot \subseteq \pi(\top\!\!\top) \subseteq \pi(\sigma(\bot\!\!\bot)) \subseteq \pi(\sigma(\pi(\top\!\!\top))) \subseteq \ldots \subseteq \ldots \subseteq \sigma(\pi(\sigma(\bot\!\!\bot))) \subseteq \sigma(\pi(\top\!\!\top)) \subseteq \sigma(\bot\!\!\bot) \subseteq \top\!\!\top.$$

It is explicitly given here, and we observe equalities in an alternating pattern:

$$\bot\!\!\bot \subseteq \overline{B;\top\!\!\top} = \overline{B;\overline{B;\bot\!\!\bot}} \subseteq \overline{B;\overline{B;\overline{B;\top\!\!\top}}} = \ldots \subseteq \ldots \subseteq \overline{B;\overline{B;\overline{B;\bot\!\!\bot}}} = \overline{B;\overline{B;\top\!\!\top}} \subseteq \overline{B;\bot\!\!\bot} = \top\!\!\top.$$

Again, the final situation is characterised by the formulae $a = \pi(b)$ and $\sigma(a) = b$, which this time turn out to be $a = \overline{B;b}$ and $\overline{B;a} = b$. In addition, we will always have $a \subseteq b$. The smaller set a gives loss positions, while the larger one then indicates win positions as $\overline{b}$ and draw positions as $b \cap \overline{a}$. This is visualized by the following diagram for sets of win, loss, and draw, the arrows of which indicate moves that must exist, may exist, or are not allowed to exist.



Schema of a game solution

A result will be found for all homogeneous relations. Often, however, all vertices will be qualified as draw. The set of draw positions may also be empty as in the solution of our initial game example.

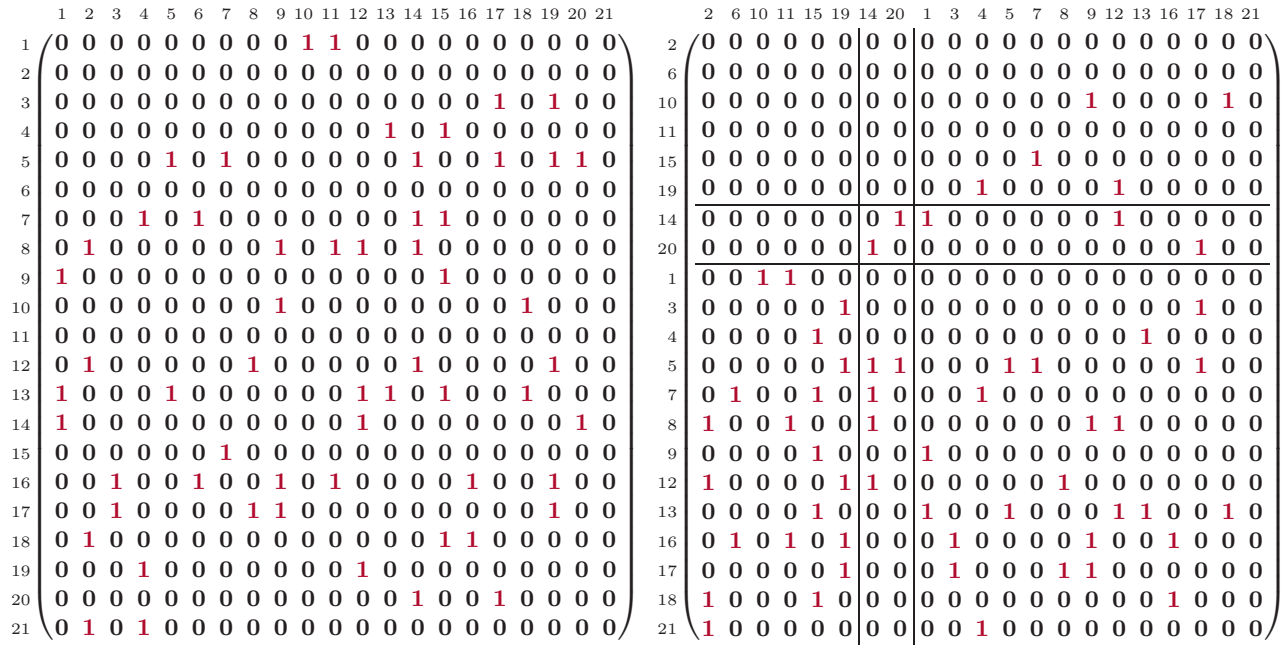
$$\begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array}\!\!\begin{array}{c} \begin{array}{ccccccc} 0 & 1 & 2 & 3 & 4 & 5 & 6 \end{array} \\ \begin{pmatrix} 0&0&0&0&0&0&0 \\ 1&0&0&0&0&0&0 \\ 1&1&0&0&0&0&0 \\ 0&1&1&0&0&0&0 \\ 0&0&1&1&0&0&0 \\ 0&0&0&1&1&0&0 \\ 0&0&0&0&1&1&0 \end{pmatrix} \end{array}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

Solution of the NIM-type game

The full power of this approach, however, will only be seen when we assign the two players different and heterogeneous relations $B : V \leftrightarrow W$ and $B' : W \leftrightarrow V$ to follow.

We now try to visualize the results of this game analysis by concentrating on the subdivision of the matrix B and the vectors a, b, respectively.

```
printResMatrixGame m =
   let (aa, bb)     = gameSolution m
       lossVector   = (head (transpMat aa))
       lossPlusDraw = (head (transpMat bb))
       drawVector   = zipWith (\ x y -> y && (not x)) lossVector lossPlusDraw
       winVector    = map not lossPlusDraw
       subdivision  = [lossVector, drawVector, winVector]
       gameRearr    = stringForNamedMatrixLines $
                        rearrangeMatWithLines m subdivision subdivision
   in  stringForOriginalNamedMatrix m ++ gameRearr
```



A random relation and its game solution rearranged

6.3.1 Proposition (*Rearranging relations with respect to a game*)**.** Every finite homogeneous relation may by simultaneously permuting rows and columns be transformed into a matrix satisfying the following basic structure with square diagonal entries:

$$\begin{pmatrix} \text{⫫} & \text{⫫} & * \\ \text{⫫} & \text{total} & * \\ \text{total} & * & * \end{pmatrix}$$

□

The subdivision into groups loss/draw/win is uniquely determined, and indeed

$$a = \begin{pmatrix} \text{⫪} \\ \text{⫫} \\ \text{⫫} \end{pmatrix} = \overline{\begin{pmatrix} \text{⫫} & \text{⫫} & * \\ \text{⫫} & \text{total} & * \\ \text{total} & * & * \end{pmatrix}; \begin{pmatrix} \text{⫪} \\ \text{⫪} \\ \text{⫫} \end{pmatrix}} \qquad b = \begin{pmatrix} \text{⫪} \\ \text{⫪} \\ \text{⫫} \end{pmatrix} = \overline{\begin{pmatrix} \text{⫫} & \text{⫫} & * \\ \text{⫫} & \text{total} & * \\ \text{total} & * & * \end{pmatrix}; \begin{pmatrix} \text{⫪} \\ \text{⫫} \\ \text{⫫} \end{pmatrix}}$$

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?

6.4 Matching and Assignment

An additional antimorphism situation is known to exist in connection with matchings and assignments. 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

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

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.

6.4.1 Definition. i) Given a possibly heterogeneous relation Q, we call λ a **Q-matching** provided it is a univalent and injective relation 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 serves to bookkeeping 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
trivialMatch q = trivialMatchAbove q (nullMatFor q)
```

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, $v \mapsto \sigma(v) = \overline{Q^\mathsf{T} ; v}$. The second shall present the set of boys not assigned to some set of girls, $w \mapsto \pi(w) = \overline{\lambda ; w}$. In Haskell, they may both be formulated applying

```
antitoneMapAssign b x = negaMat (b *** x)
```

Using these antitone mappings, we execute the standard Galois iteration, i.e., we apply the following function, which may be started in two ways.

```
data LR = LeftRight | RightLeft    deriving (Eq, Show)
assignSolution q lambda lr
      = let lv = case lr of
                   LeftRight -> startVector False q
                   RightLeft -> startVector True  q
            rv = case lr of
                   LeftRight -> startVector True  (transpMat q)
                   RightLeft -> startVector False (transpMat q)
        in  untilGS (antitoneMapAssign (transpMat q))
                    (antitoneMapAssign lambda)
                    (lv, rv)
```

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

Remembering Prop. 4.3.3, we learn that the pair a, b is an inclusion-maximal pair of independent sets for Q, or else: $\overline{a}, \overline{b}$ is an inclusion-minimal line covering.

As of yet, a, b need not be an inclusion-maximal pair of independent sets for λ, nor need $\overline{a}, \overline{b}$ be an inclusion-minimal line covering for λ! This will only be the case, when in addition $\overline{b} = \lambda^\intercal \backslash a$. We provide functions to test for these properties.

```
aBarEQUALSlamB q lambda (a, b) = negaMat a === (lambda *** b)
bBarEQUALSqTa  q lambda (a, b) = negaMat b === ((transpMat q) *** a)
aBarEQUALSQb   q lambda (a, b) = negaMat a === (q *** b)
bBarEQUALSlTa  q lambda (a, b) = negaMat b === ((transpMat lambda) *** a)
isInclMinLineCoveringForQ      q lambda (a, b) = aBarEQUALSQb   q lambda (a, b) &&
                                                 bBarEQUALSqTa  q lambda (a, b)
isInclMinLineCoveringForLambda q lambda (a, b) = aBarEQUALSlamB q lambda (a, b) &&
                                                 bBarEQUALSlTa  q lambda (a, b)
```

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 $a = \{2, 4, 6, 1, 3\}$ and $b = \{5, 3, 2\}$.

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

Sympathy and matching rearranged

We now discuss whether we had been right in deciding for the variant `LeftRight` of the iteration procedure. Assume now we had decided the other way round and had chosen to start with `RightLeft`. This would obviously mean the same as starting with `LeftRight` for $Q^\intercal$ and $\lambda^\intercal$. One will observe easily that again four conditions would be valid at the end of the iteration with $Q^\intercal$ for Q and $\lambda^\intercal$ for λ

as well as, say a', b'. Then a' corresponds to b and b' corresponds to a. This means that the resulting decomposition of the matrices does *not* depend on the choice of `LeftRight/RightLeft` — as long as all four equations are 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 $\overline{b} = Q^\intercal ; a \supsetneq \lambda^\intercal ; a$.

We make use of the formula $\lambda ; \overline{S} = \lambda ; \top\!\!\top \cap \overline{\lambda ; S}$, which holds since λ is univalent; see Prop. 3.1.4.iv. The iteration ends with $\overline{b} = Q^\intercal ; a$ and $\overline{a} = \lambda ; b$. This easily expands to

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

from which the last but one becomes

$\overline{b} = Q^\intercal ; a = Q^\intercal ; \overline{\lambda ; b} = Q^\intercal ; \overline{\lambda ; \top\!\!\top \cap \overline{\lambda ; Q^\intercal ; a}} = Q^\intercal ; (\overline{\lambda ; \top\!\!\top} \cup \lambda ; Q^\intercal ; a) = Q^\intercal ; (\overline{\lambda ; \top\!\!\top} \cup \lambda ; Q^\intercal ; (\overline{\lambda ; \top\!\!\top} \cup \lambda ; Q^\intercal ; a))$

indicating how to prove that

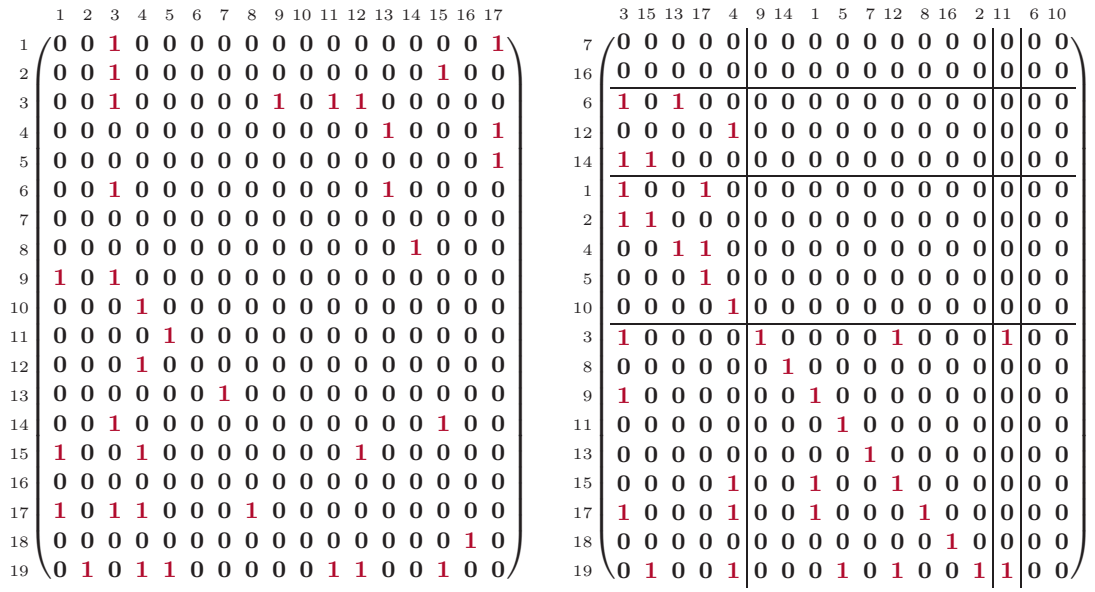
$\overline{b} = (Q^\intercal \cup Q^\intercal ; \lambda ; Q^\intercal \cup Q^\intercal ; \lambda ; Q^\intercal ; \lambda ; Q^\intercal \cup \ \dots\ ) ; \overline{\lambda ; \top\!\!\top}$

If $\lambda^\intercal ; a \subsetneq \overline{b}$, we may thus find a point in $\overline{\lambda^\intercal ; a} \cap (Q^\intercal \cup Q^\intercal ; \lambda ; Q^\intercal \cup Q^\intercal ; \lambda ; Q^\intercal ; \lambda ; Q^\intercal \cup \ \dots\ ) ; \overline{\lambda ; \top\!\!\top}$ which leads to the following alternating chain algorithm.

```
findMaxMatchFromInitial q lambda =
   let triv  = trivialMatchAbove q lambda
       (a,b) = assignSolution q triv LeftRight
       coveredFromA = transpMat lambda *** a
       test  = negaMat b === coveredFromA
       findBetterMatching q lambda  (a,b) coveredFromA =
          let qLambdaT = (q &&& (a *** [replicate (cols q) True])) *** (transpMat lambda)
              rowsMatched = map or lambda
              bBarMinusCoveredFramA = negaMat (b ||| coveredFromA)
              unMatchedInA = pointVecToNuS $ zipWith (\[x] y -> x && (not y)) a rowsMatched
              leftForUncoveredBBar = pointMatToNuS $ q *** bBarMinusCoveredFramA
              sp = shortPath qLambdaT unMatchedInA leftForUncoveredBBar
              lastRight = head $ filter (`elem` succs q (last sp))
                                        (pointMatToNuS bBarMinusCoveredFramA)
              correspondingRight = map (\x -> head $ succs lambda x) (tail sp) ++
                                   [lastRight]
          in  alternateExchange lambda sp correspondingRight
       alternateExchange lambda lLeft lRight =
          let pairList       = zip lLeft lRight
              sortedPairList = sort pairList
              rowForSecondComponent s = (replicate (s - 1)           False) ++ [True] ++
                                        (replicate (cols lambda - s) False)
              splToMatrix spl = splToMatrixAUX spl []
              splToMatrixAUX [] nm = nm ++
                       (zip (replicate (rows lambda - (length nm)) False) (repeat []))
              splToMatrixAUX ((a,b):rest) nm =
                   splToMatrixAUX rest (nm ++ (zip (replicate (a-(length nm)-1) False)
                                                   (repeat []))
                                           ++ [(True, rowForSecondComponent b)])
              zipFunction row (bool,newRow) = case bool of
                                                True  -> newRow
                                                False -> row
          in  zipWith zipFunction lambda (splToMatrix sortedPairList)
   in case test of
         True  -> triv
         False -> findMaxMatchFromInitial q (findBetterMatching q triv (a,b) coveredFromA)
```

When showing the result, some additional care will be taken concerning empty rows or columns in Q. To obtain the subdivided relations neatly, these are placed at the beginning, or at the end, depending on f being assigned the value `True` or `False`, respectively,

```
printResMatching f q lambda =
   let (a, b) = assignSolution q lambda LeftRight
       u = head $ transpMat a
       v = head $ transpMat b
       aSubdivision =
          case f of
             True  ->
              let qEmptyRow = map not (map or q)
                  matchPartnerInA = map or (lambda *** (negaMat b))
                  restInA = zipWith (&&) u
                            (map not (zipWith (||) matchPartnerInA qEmptyRow ))
              in  reverse $ filter or [map not u, matchPartnerInA, restInA, qEmptyRow ]
             False -> [map not u, u]
       bSubdivision =
          case f of
             True  -> let qEmptyCol = map not (map or (transpMat q))
                          matchPartnerInB = map or ((transpMat lambda) *** (negaMat a))
                          restInB = zipWith (&&) v
                            (map not (zipWith (||) matchPartnerInB qEmptyCol))
                      in filter or [map not v, matchPartnerInB, restInB, qEmptyCol]
             False -> [map not v, v]
       xxx = lambda &&& (negaMat a *** [replicate (cols q) True])
       yyy = transpMat $ filter or (transpMat (filter or xxx))     -- is a permutation!
       cycYYY = permMatToCyc yyy
       lBBar = length $ filter (\[x] -> not x) b
       cyc1  = map (\y -> [y]) [1..lBBar]   ++
               map (map (\z -> z + lBBar)) cycYYY ++
               map (\y -> [y]) [1 + lBBar + (length $ concat cycYYY)..cols q]
       pM  = ident $ cols q   --permCycToMat cyc1
       pMT = transpMat pM
       sympDiv  = stringForNamedMatrixLines $
                      rearrangeMatWithLines (q)      aSubdivision (bSubdivision)
       matchDiv = stringForNamedMatrixLines $
                      rearrangeMatWithLines (lambda) aSubdivision (bSubdivision)
       bBarEQUALSLAMBDATa =
           case bBarEQUALSlTa  q lambda (a, b) of
              True  -> "satisfies $\\RELneg{b}=\\lambda\\RELtraOP\\RELcompOP a$\n\n"
              False -> "violates  $\\RELneg{b}=\\lambda\\RELtraOP\\RELcompOP a$\n\n"
   in  ((stringForOriginalNamedMatrix q) ++
            (stringForOriginalNamedMatrix lambda) ++ "\n\n" ++
            sympDiv ++ matchDiv ++ "\n\n" ++ bBarEQUALSLAMBDATa)
printResMatchingFromScratch q =
   printResMatching False q (trivialMatch q)
printResMatchingFromScratchOPT q =
   printResMatching False q (findMaxMatchFromInitial q (trivialMatch q))
printResMatchingFromScratchOPTFine q =
   printResMatching True q (findMaxMatchFromInitial q (trivialMatch q))
```



Arbitrary relation with a rearrangement according to a cardinality-maximum matching — the diagonals

6.4.2 Proposition (*Decomposing according to matching and assignment*)**.** i) Any given heterogeneous relation can by independently permuting rows and columns be transformed into a matrix of the following form: It has a 2 by 2 pattern with not necessarily square diagonal blocks.

$$Q = \begin{pmatrix} \text{Hall}^\intercal & \between \\ * & \text{Hall} \end{pmatrix} \qquad \lambda = \begin{pmatrix} \text{univalent + surjective + injective} & \between \\ \between & \text{univalent + total + injective} \end{pmatrix}$$

ii) In addition, any maximum matching λ of Q will obey the subdivision on the right. The respective second rows and columns may be further decomposed in this refinement, however, depending on the maximum matching λ:

$$\left(\begin{array}{c|ccc} \between & \between & \between & \between \\ \text{total} & \between & \between & \between \\ \text{Hall}^\intercal \text{ + square} & \between & \between & \between \\ \hline * & \multicolumn{3}{c}{\text{Hall + square} \quad \text{surjective} \quad \between} \end{array}\right) \qquad \left(\begin{array}{c|ccc} \between & \between & \between & \between \\ \between & \between & \between & \between \\ \text{permutation} & \between & \between & \between \\ \hline \between & \multicolumn{3}{c}{\text{permutation} \quad \between \quad \between} \end{array}\right)$$

Here, any vanishing row of Q, if it occurs, is positioned in the first zone. Vanishing columns are shown last. The chosen λ determines necessarily square permutations inside the positions of Q that are Hall or $\text{Hall}^\intercal$, respectively. Other λ's might make another choice from the second zone and the third; correspondingly for columns. □

Looking back to Def. 4.3.1, we see that we have not just decomposed according to a pair of independent sets, but in addition ordered the rest of the relation so as to obtain matchings. This improves the visualizations shown earlier.

6.5 König's Theorems

We will now put concepts together and obtain and visualize results based on famous combinatorial concepts. The first are on line-coverings and assignments. Some sort of counting comes into play,

however, here in its algebraic form. Permutations allow 1:1-comparison of sets. Often this means to transfer heterogeneous concepts to the $n \times n$-case.

We first consider a matching (or an assignment) which is maximal with respect to cardinality. An easy observation leads to the following

6.5.1 Proposition. Let some relation Q be given together with a matching $\lambda \subseteq Q$ and the results a, b of the iteration. Then the following hold

i) $(\overline{a}, \overline{b})$ forms a line-covering and $Q \subseteq \overline{a} \mathbb{T} \cup \mathbb{T} \overline{b}^\mathsf{T}$.

ii) term rank$(Q) \leq |\overline{a}| + |\overline{b}|$

iii) $|\lambda| \leq$ term rank(Q)

iv) If $\overline{b} = \lambda^\mathsf{T} a$, then term rank$(Q) = |\overline{a}| + |\overline{b}| = |\lambda|$

Proof: i) We consider two parts of Q separately, starting with $\overline{a} \mathbb{T} \cap Q \subseteq \overline{a} \mathbb{T}$. Then, we have $\overline{b} = Q^\mathsf{T} a$ as a result of the iteration, so that

$$a \mathbb{T} \cap Q \subseteq (a \cap Q \mathbb{T}^\mathsf{T}) (\mathbb{T} \cap a^\mathsf{T} Q) \subseteq \mathbb{T} a^\mathsf{T} Q = \mathbb{T} \overline{b}^\mathsf{T}$$

ii) According to (i), the rows of $\overline{a}$ together with the columns of $\overline{b}$ cover all of Q, so that the term rank cannot be strictly above the sum of the cardinalities.

iii) A line-covering of a matching λ can obviously not be achieved with less than $|\lambda|$ lines. The matching properties $\lambda^\mathsf{T} \lambda \subseteq \mathbb{I}$, $\lambda \lambda^\mathsf{T} \subseteq \mathbb{I}$ of univalency and injectivity require that every entry of λ be covered by a separate line.

iv) Condition $\overline{b} = \lambda^\mathsf{T} a$ together with $\overline{a} = \lambda b$ shows that $|\overline{b}|$ entries of λ are needed to end in $\overline{b}$ and $|\overline{a}|$ to start in $\overline{a}$. According to injectivity no entry of λ will start in $\overline{a}$ and end in $\overline{b}$ as $\lambda \overline{b} = \lambda \lambda^\mathsf{T} a \subseteq a$. Therefore, $|\overline{a}| + |\overline{b}| \leq |\lambda|$, which in combination with (ii,iii) leads to equality. □

The following is an easy consequence which sometimes is formulated directly as a result.

6.5.2 Corollary (*König-Egervary-Theorem*)**.** For an arbitrary heterogeneous relation we have that the maximum cardinality of a matching equals the minimum cardinality of a line-covering. □

We now specialize to the homogeneous case and investigate what happens when the term rank of a homogeneous relation is less than n.

6.5.3 Proposition (*Frobenius-König; see [BR96] 2.1.4*)**.** For a finite homogeneous relation Q the following are equivalent:

i) None of all permutations P satisfies $P \subseteq Q$.

ii) There exists a pair of sets a, b such that $\overline{a}, \overline{b}$ is a line-covering with $|\overline{a}| + |\overline{b}| < n$, or equivalently, a, b is an independent pair of sets with $n < |a| + |b|$, or equivalently term rank $< n$.

iii) There exists a vector z together with a permutation P such that $Q^\mathsf{T} z \subsetneqq P z$. In other words: There exists a subset z which is mapped by Q onto a set with strictly fewer elements than z.

Proof: (i) $\Longrightarrow$ (ii): Find a maximum cardinality matching $\lambda \subseteq Q$ and execute the assignment iteration on page 48. It will end in a, b with $|\overline{a}| + |\overline{b}| = |\lambda| < n$ as λ can by assumption not be a permutation.

(ii) $\Longrightarrow$ (iii): Take $z := a$, which satisfies $Q^\mathsf{T};a = \overline{b}$ according to the iteration. Then $|\overline{b}| = n - |b| < |a|$.

(iii) $\Longrightarrow$ (i): When (iii) holds, Q violates the Hall condition for the subset z. If there were a permutation P contained in Q, we would have $Q^\mathsf{T};z \supseteq P^\mathsf{T};z$, and thus $|Q^\mathsf{T};z| \geq |P^\mathsf{T};z| = |z|$, a contradiction. □

6.6 Full Indecomposability

We now investigate a special case where term rank is n exhibiting a concept closely related to irreducibility. It takes into account not just a set and its complement as for irreducibility. Originally these investigations were triggered mainly from numerical mathematics as often computations were heavily dependent on the relational structure of coefficients being $\neq 0$ or $= 0$ and not so much on the values of the coefficients of a matrix.

6.6.1 Definition. A homogeneous relation A is called **partly decomposable** if there exists a vector $\mathbb{L} \neq x \neq \mathbb{T}$ together with a permutation P such that $A;x \subseteq P;x$. A relation A is **fully indecomposable**, if it is not partly decomposable. □

Choosing a permutation P' that sends all the $\mathbf{1}$-coefficients of x to the end, and choosing in addition a permutation P'' that sends all the $\mathbf{1}$-coefficients of $P;x$ to the end, we arrive at a rearranged form with square diagonal blocks like

$$P'';A;P'^\mathsf{T} = \begin{pmatrix} * & \mathbb{L} \\ * & * \end{pmatrix} \qquad P';x = \begin{pmatrix} \mathbb{L} \\ \mathbb{T} \end{pmatrix} \qquad P'';P;x = \begin{pmatrix} \mathbb{L} \\ \mathbb{T} \end{pmatrix}$$

Of course, every reducible relation is partly decomposable. On the other hand, being fully indecomposable is stronger than being irreducible. Given any vector $\mathbb{L} \neq x \neq \mathbb{T}$, we have for all permutations P that $A;x \not\subseteq P;x$ if A is fully indecomposable. As every permutation is tested, this is a mere counting argument meaning that the number of entries $\mathbf{1}$ in x doesn't suffice to cover the number of coefficients $\mathbf{1}$ in $A;x$, or that for any x — excluding the trivial cases $x = \mathbb{L}$ and $x = \mathbb{T}$ — $A;x$ has strictly more $\mathbf{1}$'s than x.

There is also a connection with the concepts of line-covering and independence. If A is partly decomposable based on x, P, it admits a line-covering $(P;x, \overline{x})$ and a pair $(\overline{P;x}, x)$ of independent sets. In both cases, the term rank, i.e., the sum of cardinalities, is n.

The next two propositions show properties which a fully indecomposable relation enjoys in addition to those of an irreducible relation.

6.6.2 Proposition (*See, e.g., [BR96] 2.2.1*)**.** A fully indecomposable $n \times n$-relation A satisfies $A^{n-1} = \mathbb{T}$.

Proof: The counting argument holds in particular, when we choose columns of A^k for x. As long as they are neither $\mathbb{L}$ nor $\mathbb{T}$, the number of $\mathbf{1}$'s will strictly increase in columns of A^k, where $k = 1, 2, \ldots$. Therefore, finally $A^{n-1} = A^+ = \mathbb{T}$. □

One might be tempted to compare primitive irreducible relations with fully indecomposable ones. Of course, the latter implies the former following Prop. 6.6.2 in combination with Def. 5.2.3. The so-called *index of primitivity* is defined to be the least q for which $A^q = \top\!\top$. While Prop. 6.6.2 shows that $q \leq n-1$ for fully indecomposable relations, estimations in [HV58] show only $q \leq (n-1)^2+1$ for a primitive relation.

6.6.3 Theorem (*See, e.g., [BR96] 2.2.2*)**.** If A, B are fully indecomposable then so is $A;B$.

Proof: We use the same argument as in the preceding proof. As A, B are fully indecomposable, we have for all vectors $\bot\!\!\!\bot \neq x \neq \top\!\top$ that $A;x$ as well as $B;x$ have strictly more $\mathbf{1}$'s than x. This obviously holds also for $A;B;x$. □

We now ask for the properties a fully indecomposable relation has in addition to its term rank being n. The identity, e.g., has term rank n but can be decomposed in many ways.

6.6.4 Theorem (*See, e.g., [BR96] 2.2.4*)**.** A homogeneous relation A is fully indecomposable precisely when it is the union of permutation matrices and it is chainable.

Proof: We first prove that any entry $\mathbf{1}$ of a fully irreducible relation A is also an entry in some permutation P contained in A. Consider the matrix obtained by cutting out the row and the column corresponding to the entry in question. Either it contains a permutation in which case we are done. Or it doesn't. Then the submatrix will admit a vector z as in the preceding Frobenius-König-Theorem 6.5.3 (iii) mapped by A onto a set strictly smaller in cardinality than z. Then z will in the original matrix be a set that is mapped by A onto a set with not strictly more elements and thus partly decompose A.

A will according to Prop. 4.3.6 either be chainable or admit a non-trivial line-covering $A;\overline{b} \subseteq a$. But then $A^\mathsf{T};\overline{a} \subseteq b$ and so $\overline{a}$ is mapped into b. However, we know that the term rank is n, so that $|\overline{a}|+|\overline{b}| \geq n$, and $|b| \leq |\overline{a}|$. We would have found, thus, a set that is mapped by A on a set not strictly larger in size, a contradiction to being fully indecomposable.

Next, we prove the reverse direction showing that A cannot be chainable if it is the union of permutations and it is assumed not to be fully indecomposable, i.e. there exists $A;x \subseteq P;x$, which also means $A \subseteq \overline{\overline{P;x};x^\mathsf{T}}$.

Let $A = \mathtt{sup}_{\iota\in J} P_\iota$ be a representation of A as a union of permutations. From the assumption $A;x \subseteq P;x$ we deduce $P_\iota;x \subseteq P;x$ for all $\iota \in J$.

Now we use that P as well as P_ι are permutations obtaining $P^\mathsf{T};P_\iota;x \subseteq x$ as well as $(P^\mathsf{T};P_\iota)^k;x \subseteq x$ for all k. As $P^\mathsf{T};P_\iota$ is a permutation, some power of it will equal its transpose $(P^\mathsf{T};P_\iota)^k = (P^\mathsf{T};P_\iota)^\mathsf{T}$, so that we have $(P^\mathsf{T};P_\iota)^\mathsf{T};x \subseteq x$ and consequently $P_\iota^\mathsf{T};P;x \subseteq x$ or $P;x \subseteq P_\iota;x$, i.e., finally $P_\iota;x = P;x$.

Altogether, we have $P_\iota \subseteq \mathtt{syq}((P;x)^\mathsf{T}, x^\mathsf{T})$ for all $\iota \in J$. Therefore, $A \subseteq \mathtt{syq}((P;x)^\mathsf{T}, x^\mathsf{T})$ and

$$\begin{aligned} A;A^\mathsf{T};A &\subseteq \mathtt{syq}((P;x)^\mathsf{T}, x^\mathsf{T});\mathtt{syq}(x^\mathsf{T}, (P;x)^\mathsf{T});\mathtt{syq}((P;x)^\mathsf{T}, x^\mathsf{T}) \\ &= \mathtt{syq}((P;x)^\mathsf{T}, (P;x)^\mathsf{T});\mathtt{syq}((P;x)^\mathsf{T}, x^\mathsf{T}) \\ &= \mathtt{syq}((P;x)^\mathsf{T}, x^\mathsf{T}) \\ &\subseteq \overline{\overline{P;x};x^\mathsf{T}} \end{aligned}$$

showing the same decomposition for $A;A^\mathsf{T};A$ as for A. So $A;A^\mathsf{T};A \neq \mathbb{T}$ and also $h_{\mathrm{difu}}(A) \neq \mathbb{T}$. □

The result has been known for quite a while. It was here given a relation-algebraic proof. There is a correspondence to a result on doubly-stochastic matrices. By definition, these are matrices with non-negative entries such that all row and column sums are equal to 1. The theorem states that such a matrix may always be written as a positive linear combination of permutation matrices.

7 Theory Extraction

Now, we try to automatically develop the *theory* combined with some given relations, starting from a given ontology (difunctionality, game, e.g.). When applying a relational decomposition, this theory will change as it afterwards fits into the given ontology and, thus, includes the necessary predicates and theorems.

We first present a language allowing to talk on elements, vectors (or sets), and (binary) relations together with a typing scheme. Then 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 a bit difficult.

A method of translating relational formulae to formulae in classical predicate logic is added. Then two theory extractions are shown in some detail.

7.1 Language

From the very beginning, we work in a typed or heterogeneous setting, which means that we work in a category which later stays the same as an extended theory is extracted guided by some ontology. For this category, we already provide some object names for testing purposes. Normally, however, we will be able to give names to the category objects.

```
data CatObject = O1 | O2 | O3 | O4 | Obj String          deriving (Eq, Show, Read, Ord)
```

One usually needs 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 predicates, which we call relations. A relational constant is nothing more than a name, the string, together with the types/objects 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            deriving (Eq, Ord, Read, Show)
data VectConst = Vect String CatObject            deriving (Eq, Ord, Read, Show)
data RelaConst = Rela String CatObject CatObject  deriving (Eq, Ord, Read, Show)
```

On all this, we now build first-order predicate logic, introducing individual variables, terms, and formulae. 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. An interpretation need not be given as it is generated automatically.

```
data ElemVari = Vari String CatObject                   deriving (Eq, Ord, Read, Show)
data ElemTerm = IndVar ElemVari | IndConst ElemConst    deriving (Eq, Ord, Read, Show)
data VectTerm = VCT VectConst | RelaTerm :****: VectTerm |
                VectTerm :||||: VectTerm | VectTerm :&&&&: VectTerm |
```

```
                NegaVect VectTerm | NullForV VectTerm | UnivForV VectTerm |
                PointVector ElemTerm                   deriving (Eq, Ord, Read, Show)
data RelaTerm = RCT RelaConst | RelaTerm :***: RelaTerm | RelaTerm :|||: RelaTerm |
                RelaTerm :&&&: RelaTerm | NegaRela RelaTerm | LIdent RelaTerm |
                RIdent RelaTerm | NullForR RelaTerm | UnivForR RelaTerm |
                Transp RelaTerm | VectTerm :||--: VectTerm |
                PointDiag ElemTerm                     deriving (Eq, Ord, Read, Show)
```

The typical checks are provided for well-formedness and type control.

```
elemTermIsWellFormed :: ElemTerm -> Bool
elemTermIsWellFormed e =
    case e of
       IndVar   (Vari _ _) -> True
       IndConst (Elem _ _) -> True

syntMaterialUsedInElemTerm e =
    case e of
       IndVar   ev -> ([ev],  [],[],[])
       IndConst ec -> (  [],[ec],[],[])

vectTermIsWellFormed :: VectTerm -> Bool
vectTermIsWellFormed vt =
   case vt of
      VCT _           -> True
      rt  :****: vt   -> let (o1,o2) = typeOfRelaTerm rt
                         in  relaTermIsWellFormed rt  && vectTermIsWellFormed vt &&
                             (o2 == typeOfVectTerm vt)
      vt1 :||||: vt2 -> vectTermIsWellFormed vt1 && vectTermIsWellFormed vt2 &&
                         (typeOfVectTerm vt1 == typeOfVectTerm vt2)
      vt1 :&&&&: vt2 -> vectTermIsWellFormed vt1 && vectTermIsWellFormed vt2 &&
                         (typeOfVectTerm vt1 == typeOfVectTerm vt2)
      NegaVect vt1   -> vectTermIsWellFormed vt1
      NullForV vt1   -> vectTermIsWellFormed vt1
      UnivForV vt1   -> vectTermIsWellFormed vt1
      PointVector et -> elemTermIsWellFormed et

putSyntMatTogether (a,b,c,d) (u,v,w,x) =
   (nub $ a ++ u, nub $ b ++ v, nub $ c ++ w, nub $ d ++ x)

syntMaterialUsedInVectTerm vt =
   case vt of
      VCT vc          -> ([],[],[vc],[])
      rt  :****: vt   -> let (a,b,c,d) = syntMaterialUsedInRelaTerm rt
                             (u,v,w,x) = syntMaterialUsedInVectTerm vt
                         in  putSyntMatTogether (a,b,c,d) (u,v,w,x)
      vt1 :||||: vt2 -> let (a,b,c,d) = syntMaterialUsedInVectTerm vt1
                             (u,v,w,x) = syntMaterialUsedInVectTerm vt2
                         in  putSyntMatTogether (a,b,c,d) (u,v,w,x)
      vt1 :&&&&: vt2 -> let (a,b,c,d) = syntMaterialUsedInVectTerm vt1
                             (u,v,w,x) = syntMaterialUsedInVectTerm vt2
                         in  putSyntMatTogether (a,b,c,d) (u,v,w,x)
      NegaVect vt1   -> syntMaterialUsedInVectTerm vt1
      NullForV vt1   -> syntMaterialUsedInVectTerm vt1
      UnivForV vt1   -> syntMaterialUsedInVectTerm vt1
      PointVector et -> syntMaterialUsedInElemTerm et
```

```
relaTermIsWellFormed :: RelaTerm -> Bool
relaTermIsWellFormed rt =
   case rt of
      RCT _              -> True
      rt1 :***: rt2      -> relaTermIsWellFormed rt1 && relaTermIsWellFormed rt2 &&
                            ((snd $ typeOfRelaTerm rt1) == (fst $ typeOfRelaTerm rt2))
      rt1 :|||: rt2      -> relaTermIsWellFormed rt1 && relaTermIsWellFormed rt2 &&
                            (typeOfRelaTerm rt1 == typeOfRelaTerm rt2)
      rt1 :&&&: rt2      -> relaTermIsWellFormed rt1 && relaTermIsWellFormed rt2 &&
                            (typeOfRelaTerm rt1 == typeOfRelaTerm rt2)
      NegaRela  rt1      -> relaTermIsWellFormed rt1
      LIdent    rt1      -> relaTermIsWellFormed rt1
      RIdent    rt1      -> relaTermIsWellFormed rt1
      NullForR  rt1      -> relaTermIsWellFormed rt1
      UnivForR  rt1      -> relaTermIsWellFormed rt1
      Transp    rt1      -> relaTermIsWellFormed rt1
      vt1 :||--: vt2     -> vectTermIsWellFormed vt1 && vectTermIsWellFormed vt2
      PointDiag et       -> elemTermIsWellFormed et

syntMaterialUsedInRelaTerm rt =
   case rt of
      RCT rc             -> ([],[],[],[rc])
      rt1 :***: rt2      -> let (a,b,c,d) = syntMaterialUsedInRelaTerm rt1
                                (u,v,w,x) = syntMaterialUsedInRelaTerm rt2
                            in  putSyntMatTogether (a,b,c,d) (u,v,w,x)
      rt1 :|||: rt2      -> let (a,b,c,d) = syntMaterialUsedInRelaTerm rt1
                                (u,v,w,x) = syntMaterialUsedInRelaTerm rt2
                            in  putSyntMatTogether (a,b,c,d) (u,v,w,x)
      rt1 :&&&: rt2      -> let (a,b,c,d) = syntMaterialUsedInRelaTerm rt1
                                (u,v,w,x) = syntMaterialUsedInRelaTerm rt2
                            in  putSyntMatTogether (a,b,c,d) (u,v,w,x)
      NegaRela  rt1      -> syntMaterialUsedInRelaTerm rt1
      LIdent    rt1      -> syntMaterialUsedInRelaTerm rt1
      RIdent    rt1      -> syntMaterialUsedInRelaTerm rt1
      NullForR  rt1      -> syntMaterialUsedInRelaTerm rt1
      UnivForR  rt1      -> syntMaterialUsedInRelaTerm rt1
      Transp    rt1      -> syntMaterialUsedInRelaTerm rt1
      vt1 :||--: vt2     -> let (a,b,c,d) = syntMaterialUsedInVectTerm vt1
                                (u,v,w,x) = syntMaterialUsedInVectTerm vt2
                            in  putSyntMatTogether (a,b,c,d) (u,v,w,x)
      PointDiag et       -> syntMaterialUsedInElemTerm et

typeOfElemTerm :: ElemTerm -> CatObject
typeOfElemTerm t =
    case t of
       IndVar   (Vari _ o) -> o
       IndConst (Elem _ o) -> o

typeOfVectTerm :: VectTerm -> CatObject
typeOfVectTerm vt =
    case vt of
       VCT (Vect  _ o) -> o
       rt  :****: vt2  -> fst $ typeOfRelaTerm rt
       vt1 :||||: vt2  -> typeOfVectTerm vt1
       vt1 :&&&&: vt2  -> typeOfVectTerm vt1
```

```
       NegaVect   vt1  -> typeOfVectTerm vt1
       NullForV   vt1  -> typeOfVectTerm vt1
       UnivForV   vt1  -> typeOfVectTerm vt1
       PointVector et  -> typeOfElemTerm et

typeOfRelaTerm :: RelaTerm -> (CatObject,CatObject)
typeOfRelaTerm rt =
    case rt of
       RCT (Rela _ o o') -> (o,o')
       rt1 :***: rt2     -> (fst $ typeOfRelaTerm rt1, snd $ typeOfRelaTerm rt2)
       rt1  :|||: _      -> typeOfRelaTerm rt1
       rt1  :&&&: _      -> typeOfRelaTerm rt1
       NegaRela rt1      -> typeOfRelaTerm rt1
       LIdent   rt1      -> (a,a)    where a = fst $ typeOfRelaTerm rt1
       RIdent   rt1      -> (b,b)    where b = snd $ typeOfRelaTerm rt1
       NullForR rt1      -> typeOfRelaTerm rt1
       UnivForR rt1      -> typeOfRelaTerm rt1
       Transp   rt1      -> (b,a)    where (a,b) =   typeOfRelaTerm rt1
       vt1 :||--: vt2    -> (typeOfVectTerm vt1,typeOfVectTerm vt2)
       PointDiag et      -> (a,a)    where a = typeOfElemTerm et
```

Three forms of formulae are distinguished in order to facilitate maintenance of ubiquitous typing.

```
data ElemForm = Verum | Falsum | VC VectTerm ElemTerm | PC RelaTerm ElemTerm ElemTerm |
                Equation ElemTerm ElemTerm | Negated ElemForm | Implies ElemForm ElemForm |
                Disjunct ElemForm ElemForm | Conjunct ElemForm ElemForm |
                UnivQuantForm ElemVari ElemForm | ExistQuantForm ElemVari ElemForm
                                                    deriving (Eq, Ord, Read, Show)
data VectForm = VectTerm :<===: VectTerm | VectTerm :>===: VectTerm
                                                    deriving (Eq, Ord, Read, Show)
data RelaForm = RelaTerm :<==:  RelaTerm | RelaTerm :>==:  RelaTerm
                                                    deriving (Eq, Ord, Read, Show)
```

Again, typing and well-formedness is defined as usual. The type of an element formula is intended to be `Bool`; we have, however, kept us apart from using machine- or programming language-dependent facilities as long as possible.

```
typeOfVectForm :: VectForm -> CatObject
typeOfVectForm vf =
    case vf of
      vt1 :<===: vt2 -> typeOfVectTerm vt1
      vt1 :>===: vt2 -> typeOfVectTerm vt1

typeOfRelaForm :: RelaForm -> (CatObject,CatObject)
typeOfRelaForm rf =
    case rf of
      rt1 :<==: rt2 -> typeOfRelaTerm rt1
      rt1 :>==: rt2 -> typeOfRelaTerm rt1


elemFormIsWellFormed :: ElemForm -> Bool
elemFormIsWellFormed f =
   case f of
      Verum               -> True
      Falsum              -> True
      VC    vt t          -> vectTermIsWellFormed vt && elemTermIsWellFormed t &&
```

```
                              (typeOfElemTerm t == typeOfVectTerm vt)
      PC    rt t1 t2      -> let (o1,o2) = typeOfRelaTerm rt
                              in  relaTermIsWellFormed rt &&
                                  elemTermIsWellFormed t1 && elemTermIsWellFormed t2 &&
                                  (typeOfElemTerm t1 == o1) && (typeOfElemTerm t2 == o2)
      Equation t1 t2      -> elemTermIsWellFormed t1 && elemTermIsWellFormed t2 &&
                              (typeOfElemTerm t1 == typeOfElemTerm t2)
      Negated  f1         -> elemFormIsWellFormed f1
      Implies  f1 f2      -> elemFormIsWellFormed f1 && elemFormIsWellFormed f2
      Disjunct f1 f2      -> elemFormIsWellFormed f1 && elemFormIsWellFormed f2
      Conjunct f1 f2      -> elemFormIsWellFormed f1 && elemFormIsWellFormed f2
      UnivQuantForm  v f1 -> elemFormIsWellFormed f1
      ExistQuantForm v f1 -> elemFormIsWellFormed f1

syntMaterialUsedInElemForm f =
   case f of
      Verum               -> ([],[],[],[])
      Falsum              -> ([],[],[],[])
      VC    vt t          -> let (a,b,c,d) = syntMaterialUsedInVectTerm vt
                                  (u,v,w,x) = syntMaterialUsedInElemTerm  t
                              in  putSyntMatTogether (a,b,c,d) (u,v,w,x)
      PC    rt t1 t2      -> let (a,b,c,d) = syntMaterialUsedInRelaTerm rt
                                  (u,v,w,x) = syntMaterialUsedInElemTerm t1
                                  (p,q,r,s) = syntMaterialUsedInElemTerm t2
                              in  putSyntMatTogether (a,b,c,d)
                                                     (putSyntMatTogether (u,v,w,x) (p,q,r,s))
      Equation t1 t2      -> let (a,b,c,d) = syntMaterialUsedInElemTerm t1
                                  (u,v,w,x) = syntMaterialUsedInElemTerm t2
                              in  putSyntMatTogether (a,b,c,d) (u,v,w,x)
      Negated  f1         -> syntMaterialUsedInElemForm f1
      Implies  f1 f2      -> let (a,b,c,d) = syntMaterialUsedInElemForm f1
                                  (u,v,w,x) = syntMaterialUsedInElemForm f2
                              in  putSyntMatTogether (a,b,c,d) (u,v,w,x)
      Disjunct f1 f2      -> let (a,b,c,d) = syntMaterialUsedInElemForm f1
                                  (u,v,w,x) = syntMaterialUsedInElemForm f2
                              in  putSyntMatTogether (a,b,c,d) (u,v,w,x)
      Conjunct f1 f2      -> let (a,b,c,d) = syntMaterialUsedInElemForm f1
                                  (u,v,w,x) = syntMaterialUsedInElemForm f2
                              in  putSyntMatTogether (a,b,c,d) (u,v,w,x)
      UnivQuantForm  v f1 -> syntMaterialUsedInElemForm f1
      ExistQuantForm v f1 -> syntMaterialUsedInElemForm f1

vectFormIsWellFormed :: VectForm -> Bool
vectFormIsWellFormed vf =
    case vf of
      vt1 :<===: vt2 -> vectTermIsWellFormed vt1 && vectTermIsWellFormed vt2 &&
                        (typeOfVectTerm vt1 == typeOfVectTerm vt2)
      vt1 :>===: vt2 -> vectTermIsWellFormed vt1 && vectTermIsWellFormed vt2 &&
                        (typeOfVectTerm vt1 == typeOfVectTerm vt2)

syntMaterialUsedInVectForm vf =
    case vf of
      vt1 :<===: vt2 -> let (a,b,c,d) = syntMaterialUsedInVectTerm vt1
                            (u,v,w,x) = syntMaterialUsedInVectTerm vt2
                        in  putSyntMatTogether (a,b,c,d) (u,v,w,x)
      vt1 :>===: vt2 -> let (a,b,c,d) = syntMaterialUsedInVectTerm vt1
```

```
                          (u,v,w,x) = syntMaterialUsedInVectTerm vt2
                      in  putSyntMatTogether (a,b,c,d) (u,v,w,x)

relaFormIsWellFormed :: RelaForm -> Bool
relaFormIsWellFormed rf =
    case rf of
      rt1 :<==: rt2 -> relaTermIsWellFormed rt1 && relaTermIsWellFormed rt2 &&
                       (typeOfRelaTerm rt1 == typeOfRelaTerm rt2)
      rt1 :>==: rt2 -> relaTermIsWellFormed rt1 && relaTermIsWellFormed rt2 &&
                       (typeOfRelaTerm rt1 == typeOfRelaTerm rt2)

syntMaterialUsedInRelaForm rf =
    case rf of
      rt1 :<==: rt2 -> let (a,b,c,d) = syntMaterialUsedInRelaTerm rt1
                           (u,v,w,x) = syntMaterialUsedInRelaTerm rt2
                       in  putSyntMatTogether (a,b,c,d) (u,v,w,x)
      rt1 :>==: rt2 -> let (a,b,c,d) = syntMaterialUsedInRelaTerm rt1
                           (u,v,w,x) = syntMaterialUsedInRelaTerm rt2
                       in  putSyntMatTogether (a,b,c,d) (u,v,w,x)
```

Free variables are defined. Here, they originate from individual variables which are used to define point vectors etc.

```
freeVarInElemTerm :: ElemTerm -> [ElemVari]
freeVarInElemTerm t = case t of IndVar   v -> [v]
                                IndConst _ -> []

freeVarInVectTerm :: VectTerm -> [ElemVari]
freeVarInVectTerm vt =
   case vt of
      VCT _          -> []
      rt  :****: vt2 -> nub $ freeVarInRelaTerm rt  ++ freeVarInVectTerm vt2
      vt1 :||||: vt2 -> nub $ freeVarInVectTerm vt1 ++ freeVarInVectTerm vt2
      vt1 :&&&&: vt2 -> nub $ freeVarInVectTerm vt1 ++ freeVarInVectTerm vt2
      NegaVect   vt1 -> freeVarInVectTerm vt1
      NullForV   vt1 -> []
      UnivForV   vt1 -> []
      PointVector et -> freeVarInElemTerm et

freeVarInRelaTerm :: RelaTerm -> [ElemVari]
freeVarInRelaTerm rt =
   case rt of
      RCT _          -> []
      rt1 :***: rt2  -> nub $ freeVarInRelaTerm rt1 ++ freeVarInRelaTerm rt2
      rt1 :|||: rt2  -> nub $ freeVarInRelaTerm rt1 ++ freeVarInRelaTerm rt2
      rt1 :&&&: rt2  -> nub $ freeVarInRelaTerm rt1 ++ freeVarInRelaTerm rt2
      NegaRela  rt1  -> freeVarInRelaTerm rt1
      LIdent    rt1  -> []
      RIdent    rt1  -> []
      NullForR  rt1  -> []
      UnivForR  rt1  -> []
      Transp    rt1  -> freeVarInRelaTerm rt1
      vt1 :||--: vt2 -> nub $ freeVarInVectTerm vt1 ++ freeVarInVectTerm vt2
      PointDiag et   -> freeVarInElemTerm et

freeVarInElemForm :: ElemForm -> [ElemVari]
```

```
freeVarInElemForm f =
   case f of
      Verum               -> []
      Falsum              -> []
      VC      vt t        -> nub (freeVarInElemTerm t  ++ (freeVarInVectTerm vt))
      PC      rt t1 t2    -> nub (freeVarInElemTerm t1 ++ (freeVarInElemTerm t2)
                                                       ++ (freeVarInRelaTerm rt))
      Equation   t1 t2    -> nub (freeVarInElemTerm t1 ++ (freeVarInElemTerm t2))
      Negated        f1   -> freeVarInElemForm f1
      Implies        f1 f2 -> nub (freeVarInElemForm f1 ++ (freeVarInElemForm f2))
      Conjunct       f1 f2 -> nub (freeVarInElemForm f1 ++ (freeVarInElemForm f2))
      Disjunct       f1 f2 -> nub (freeVarInElemForm f1 ++ (freeVarInElemForm f2))
      UnivQuantForm  v  f' -> filter (/= v) (freeVarInElemForm f')
      ExistQuantForm v  f' -> filter (/= v) (freeVarInElemForm f')

isClosedElemForm f = freeVarInElemForm f == []
```

7.2 Models

We will now offer the opportunity to interprete the language we have defined in a model. Via an interpretation, the objects get assigned sets in this model, however, we just mention the cardinalities of the sets as they are intended to later correspond to row and column entries. Also vector and relation denotations are assigned concrete versions by the model.

```
data InterpreteObjs = Carrier  CatObject   Int    deriving (Eq, Show, Read, Ord)
data InterpreteCons = InterCon ElemConst   Int    deriving (Eq, Show, Read, Ord)
data InterpreteVect = InterVec VectConst  [Bool]  deriving (Eq, Show, Read, Ord)
data InterpreteRela = InterRel RelaConst [[Bool]] deriving (Eq, Show, Read, Ord)
```

Only in rare cases as, e.g., studying rooted graphs with the root distinguished, will we have individual constants. We provide an automatic interpretation for null relations, universal relations, and identity relations.

Putting this together, a model defined as follows:

```
data Model = MO String             -- name of the model
               [InterpreteObjs]
               [InterpreteCons]
               [InterpreteVect]
               [InterpreteRela] deriving (Eq, Show, Read, Ord)
```

In our exposition, the concrete relations and vectors are usually given as a starting point. Row and column entries owe their existence only to the analysis of the given relations and vectors. This means that we provide some generic mechanisms on the model side: We are able to check, whether the sets in question are assigned to objects consistently by the interpretations.

```
getObjectCarrierSize os o =
   (\(Carrier _ n)  -> n) $ head $ dropWhile (\(Carrier  c _) -> c /= o) os
getElemConstInterpretation cs c =
   (\(InterCon a b) -> b) $ head $ dropWhile (\(InterCon e f) -> c /= e) cs
```

```
getVectConstInterpretation vs v =
   (\(InterVec a b) -> b) $ head $ dropWhile (\(InterVec e f) -> v /= e) vs
getRelaConstInterpretation rs r =
   (\(InterRel a b) -> b) $ head $ dropWhile (\(InterRel e f) -> r /= e) rs

getArityCarrier   m et =
   let tyEt = typeOfElemTerm et
   in  arity m tyEt

getArityVectConst m vt =
   let tyVt = typeOfVectTerm vt
   in  arity m tyVt

getArityRelaConst m rt =
   let (a,b) = typeOfRelaTerm rt
   in  (arity m a,arity m b)
```

Lots of technicalities are necessary to ensure that this works as it is supposed to.

```
namesDisjointM (MO s os cs vs rs) =
    let elemNames = nub $ map (\(InterCon (Elem s _)   _)  -> s) cs
        vectNames = nub $ map (\(InterVec (Vect s _)   _)  -> s) vs
        relaNames = nub $ map (\(InterRel (Rela s _ _) _)  -> s) rs
        lcs = length cs
        lvs = length vs
        lrs = length rs
        cDisjoint = length elemNames == lcs
        vDisjoint = length vectNames == lvs
        rDisjoint = length relaNames == lrs
        allDisjoint = length (nub (elemNames ++ vectNames ++ relaNames)) == lcs + lvs + lrs
    in  cDisjoint && vDisjoint && rDisjoint && (s `notElem` elemNames) && allDisjoint &&
        (s `notElem` vectNames) && (s `notElem` relaNames)
aritiesConsistent (MO s os cs vs rs) =
    let ooo (Carrier          o  n)    = (o, n)
        eee (InterCon (Elem s o) n)    = (o, n)
        vvv (InterVec (Vect s o) v)    = (o, length v)
        rrr (InterRel (Rela s o o') m) = [(o, rows m), (o', cols m)]
        carrSetAndSize  = map ooo os
        elemNumbInSets  = map eee cs
        vectSetAndSize  = map vvv vs
        relaSetAndSize  = concat (map rrr rs)
        noDiscrepancies = length carrSetAndSize ==
             (length $ nub (carrSetAndSize ++ vectSetAndSize ++ relaSetAndSize))
        elementsInSet (o,e) =
             e <= (snd $ head $ dropWhile (\x -> o /= fst x) carrSetAndSize)
        allElementsInside   = and (map elementsInSet elemNumbInSets)
    in  noDiscrepancies && allElementsInside

arities (MO _ os _ _ _) =
   let ooo (Carrier o n) = (o, n)
   in  map ooo os
arity m o = snd (head $ dropWhile (\(o',n) -> o' /= o) (arities m))
```

7.3 Interpretation

Before interpretation is possible, we need valuations of the individual variables. Using a rather primitive lookup version, the rest is then standard.

```
type InterpreteVari = (ElemVari, Int)
type Valuations = [InterpreteVari]

lookupPrimitive           :: Eq a => a -> [(a,b)] -> b
lookupPrimitive k ((x,y):xys)
      | k==x      = y
      | otherwise = lookupPrimitive k xys

valuation vi v = lookupPrimitive v vi

interpreteElemTerm :: Model -> Valuations -> ElemTerm -> Int
interpreteElemTerm m vi t =
   let MO _ _ cs vs _ = m
   in  case t of
         IndConst c -> (\(InterCon a b) -> b) $ head $
                       dropWhile (\(InterCon name value) -> c /= name) cs
         IndVar   v -> valuation vi v

interpreteVectTerm :: Model -> Valuations -> VectTerm -> [Bool]
interpreteVectTerm m vi vt =
   let MO _ os _ vs _ = m
   in  case vt of
         VCT vc         -> interpreteVectConst m vi vc
         rt  :****: vt1 -> map head $ (interpreteRelaTerm  m vi rt ***
                           (map (\x -> [x]) (interpreteVectTerm m vi vt1)))
         vt1 :||||: vt2 -> zipWith (||) (interpreteVectTerm m vi vt1)
                                        (interpreteVectTerm m vi vt2)
         vt1 :&&&&: vt2 -> zipWith (&&) (interpreteVectTerm m vi vt1)
                                        (interpreteVectTerm m vi vt2)
         NegaVect vt1   -> map not $ interpreteVectTerm m vi vt1
         NullForV vt1   -> replicate (getObjectCarrierSize os $ typeOfVectTerm vt1) False
         UnivForV vt1   -> replicate (getObjectCarrierSize os $ typeOfVectTerm vt1) True
         PointVector et -> take (getObjectCarrierSize os $ typeOfElemTerm et)
                           (map (== interpreteElemTerm m vi et) [1..])

interpreteVectConst :: Model -> Valuations -> VectConst -> [Bool]
interpreteVectConst m _ vc =
   let MO _ _ _ vs _ = m
   in  getVectConstInterpretation vs vc

interpreteRelaTerm :: Model -> Valuations -> RelaTerm -> [[Bool]]
interpreteRelaTerm m vi rt =
   let MO _ _ _ _ rs = m
   in  case rt of
         RCT rc         -> interpreteRelaConst m vi rc
         rt1 :***: rt2  -> interpreteRelaTerm  m vi rt1 *** interpreteRelaTerm  m vi rt2
         rt1 :|||: rt2  -> interpreteRelaTerm  m vi rt1 ||| interpreteRelaTerm  m vi rt2
         rt1 :&&&: rt2  -> interpreteRelaTerm  m vi rt1 &&& interpreteRelaTerm  m vi rt2
         NegaRela  rt1  -> negaMat    $ interpreteRelaTerm  m vi rt1
         NullForR  rt1  -> nulMatNM p q   where (p,q) = getArityRelaConst m rt1
         UnivForR  rt1  -> allMatNM p q   where (p,q) = getArityRelaConst m rt1
         LIdent    rt1  -> ident    p     where (p,q) = getArityRelaConst m rt1
```

```
          RIdent    rt1  -> ident    q     where (p,q) = getArityRelaConst m rt1
          Transp    rt1  -> transpMat  $ interpreteRelaTerm  m vi rt1
          vt1 :||--: vt2 -> (map (\i -> [i]) (interpreteVectTerm m vi vt1)) ***
                            (map (\i -> [i]) (interpreteVectTerm m vi vt2))
          PointDiag et   -> pv *** (transpMat pv)
                                where pv = map (\i -> [i])
                                       (interpreteVectTerm m vi (PointVector et))

interpreteRelaConst :: Model -> Valuations -> RelaConst -> [[Bool]]
interpreteRelaConst m _ rc =
   let MO _ _ _ _ rs = m
   in  getRelaConstInterpretation rs rc

interpreteElemFormInModel :: Model -> Valuations -> ElemForm -> Bool
interpreteElemFormInModel m vi f =
   let MO _ os cs vs rs = m
   in  case f of
          Verum       -> True
          Falsum      -> False
          VC vt t     -> let intVT = interpreteVectTerm m vi vt
                             intT  = interpreteElemTerm m vi t
                         in  intVT !! (intT - 1)
          PC rt t1 t2 -> let intRT = interpreteRelaTerm m vi rt
                             intT1 = interpreteElemTerm m vi t1
                             intT2 = interpreteElemTerm m vi t2
                         in  intRT !! (intT1 - 1) !! (intT2 - 1)
          Equation t1 t2 -> interpreteElemTerm m vi t1 == interpreteElemTerm m vi t2
          Negated  f1    -> not (interpreteElemFormInModel m vi f1)
          Implies  f1 f2 -> not (interpreteElemFormInModel m vi f1)  ||
                                (interpreteElemFormInModel m vi f2)
          Conjunct f1 f2 ->     (interpreteElemFormInModel m vi f1)  &&
                                (interpreteElemFormInModel m vi f2)
          Disjunct f1 f2 ->     (interpreteElemFormInModel m vi f1)  ||
                                (interpreteElemFormInModel m vi f2)
          UnivQuantForm v f  ->
               let Vari s1 o1 = v
                   carriers = [1 .. getObjectCarrierSize os o1]
                   lOHNEv = filter (\(a,b) -> a /= v) vi
                   viMOD  = [ (v, nn) | nn <- carriers ]
                   vis = map (\p -> ((p : lOHNEv))) viMOD
               in  and (map (\vi' -> interpreteElemFormInModel m vi' f) vis)
          ExistQuantForm v f ->
               let Vari s1 o1 = v
                   carriers = [1 .. getObjectCarrierSize os o1]
                   lOHNEv = filter (\(a,b) -> a /= v) vi
                   viMOD  = [ (v, nn) | nn <- carriers ]
                   vis = map (\p -> ((p : lOHNEv))) viMOD
               in  or  (map (\vi' -> interpreteElemFormInModel m vi' f) vis)
interpreteVectFormInModel :: Model -> Valuations -> VectForm -> Bool
interpreteVectFormInModel m vi vf =
   case vf of
      (vt1 :<===: vt2) -> let int1 = interpreteVectTerm m vi vt1
                              int2 = interpreteVectTerm m vi vt2
                          in  int1 <= int2
      (vt1 :>===: vt2) -> let int1 = interpreteVectTerm m vi vt1
                              int2 = interpreteVectTerm m vi vt2
```

```
                          in  int2 <= int1

interpreteRelaFormInModel :: Model -> Valuations -> RelaForm -> Bool
interpreteRelaFormInModel m vi rf =
   case rf of
      (rt1 :<==: rt2) -> let int1 = interpreteRelaTerm m vi rt1
                             int2 = interpreteRelaTerm m vi rt2
                         in  int1 <== int2
      (rt1 :>==: rt2) -> let int1 = interpreteRelaTerm m vi rt1
                             int2 = interpreteRelaTerm m vi rt2
                         in  int2 <== int1
```

The following is a first model for test purposes.

```
testModel = MO "TestModel" [] [] [difuSetL0, difuSetL1, difuSetL2, difuSetL3,
                                  difuSetR1, difuSetR2, difuSetR3] [ir1, ir2]
iv1 = InterVec (Vect "set" O1)  [True, False, True]


difuSetL0 = InterVec (Vect "difuSetL0" O1)
  [False, False, False, False, False, False,
   False, True,  False, False, False, False]
difuSetL1 = InterVec (Vect "difuSetL1" O1)
  [True,  False, False, True,  False, True,
   False, False, False, True,  False, True ]
difuSetL2 = InterVec (Vect "difuSetL2" O1)
  [False, True,  False, False, True,  False,
   True,  False, False, False, False, False]
difuSetL3 = InterVec (Vect "difuSetL3" O1)
  [False, False, True,  False, False, False,
   False, False, True,  False, True,  False]

difuSetR1 = InterVec (Vect "difuSetR1" O2)
  [True,  False, True,  False, True,  False,
   False, True,  True,  False, True,  True,  False, True ]
difuSetR2 = InterVec (Vect "difuSetR2" O2)
  [False, True,  False, True,  False, False,
   True,  False, False, True,  False, False, False, False]
difuSetR3 = InterVec (Vect "difuSetR3" O2)
  [False, False, False, False, False, True,
   False, False, False, False, False, False, True,  False]


ir1 = InterRel (Rela "testDifu"  O1 O2) difuTestMatrix
ir2 = InterRel (Rela "testDifuT" O2 O1) (transpMat difuTestMatrix)

val1, val2, val3, val4 :: (ElemVari, Int)
val1 = (Vari "x" O1, 3)
val2 = (Vari "y" O2, 2)
val3 = (Vari "z" O2, 3)
val4 = (Vari "w" O1, 3)
vals = [val1, val2, val3, val4]


closedTestElemForm = ExistQuantForm (Vari "x" O1)
                                    (PC (RCT (Rela "testDifu" O1 O2) )
```

```
                                    (IndVar (Vari "x" 01))
                                    (IndVar (Vari "z" 02))
                                )
difuTestElemForm =
    UnivQuantForm
       (Vari "x" 01)
       (UnivQuantForm
          (Vari "y" 02)
          (ExistQuantForm
              (Vari "z" 02)
              (ExistQuantForm
                 (Vari "w" 01)
                 (Implies (Conjunct (Conjunct (PC (RCT (Rela "testDifu"  01 02))
                                                  (IndVar (Vari "x" 01))
                                                  (IndVar (Vari "z" 02))    )
                                              (PC (RCT (Rela "testDifuT" 02 01))
                                                  (IndVar (Vari "z" 02))
                                                  (IndVar (Vari "w" 01))    )
                                    )
                                    (PC (RCT (Rela "testDifu" 01 02))
                                        (IndVar (Vari "w" 01))
                                        (IndVar (Vari "y" 02))   )
                          )
                          (PC (RCT (Rela "testDifu" 01 02))
                              (IndVar (Vari "x" 01))
                              (IndVar (Vari "y" 02))   )
                 )
              )
          )
       )
```

7.4 Example Extraction

We learn how to formulate a theory and how to check it for being well-formulated.

```
data Theory = TH String         -- name of the theory
                 [CatObject]    -- carrier set denotations encountered in the theory
                 [ElemConst]    -- element denotations encountered in the theory
                 [VectConst]    -- subset denotations encountered in the theory
                 [RelaConst]    -- relation denotations encountered in the theory
                 [ElemForm]     -- element formulae demanded to hold
                 [VectForm]     -- vector formulae demanded to hold
                 [RelaForm]     -- relation formulae demanded to hold
     deriving (Read, Show, Eq, Ord)

checkTheoryWelldefined th =
   let TH s os cs vs rs ef vf rf = th
       objectsInElemConsts = map (\(Elem _ o)    -> o) cs
       objectsInVectConsts = map (\(Vect _ o)    -> o) vs
       objectsInRelaConsts = concat $ map (\(Rela _ so to) -> [so, to]) rs
       quadrupel1 = foldr1 putSyntMatTogether $ map syntMaterialUsedInElemForm ef
       quadrupel2 = foldr1 putSyntMatTogether $ map syntMaterialUsedInVectForm vf
       quadrupel3 = foldr1 putSyntMatTogether $ map syntMaterialUsedInRelaForm rf
       quadrupel  = putSyntMatTogether quadrupel1
```

```
                                    (putSyntMatTogether quadrupel2 quadrupel3)
       onlySyntMatFromTheory (_,b,c,d) th =
          let constsOK = and (map (`elem` cs) b)
              vectsOK  = and (map (`elem` vs) c)
              relsOK   = and (map (`elem` rs) d)
          in  constsOK && vectsOK && relsOK
   in  (sort os == (sort $ nub (objectsInElemConsts ++ objectsInVectConsts
                                                    ++ objectsInRelaConsts))) &&
       (namesDisjointT th) && (onlySyntMatFromTheory quadrupel1 th)

namesDisjointT (TH s os cs vs rs _ _ _) =
    let elemNames = nub $ map (\(Elem s _)   -> s) cs
        vectNames = nub $ map (\(Vect s _)   -> s) vs
        relaNames = nub $ map (\(Rela s _ _) -> s) rs
        lcs = length cs
        lvs = length vs
        lrs = length rs
        cDisjoint = length elemNames == lcs
        vDisjoint = length vectNames == lvs
        rDisjoint = length relaNames == lrs
        allDisjoint = length (nub (elemNames ++ vectNames ++ relaNames)) == lcs + lvs + lrs
    in  cDisjoint && vDisjoint && rDisjoint && (s `notElem` elemNames) && allDisjoint &&
        (s `notElem` vectNames) && (s `notElem` relaNames)
```

We can check whether a given model is a model for some theory.

```
checkIsModelForTheory mo th =
   let TH thS osT csT vsT rsT ef vf rf = th
       MO moS osM csM vsM rsM = mo
       elemFormsSatisfied = and $ map (interpreteElemFormInModel mo []) ef
       vectFormsSatisfied = and $ map (interpreteVectFormInModel mo []) vf
       relaFormsSatisfied = and $ map (interpreteRelaFormInModel mo []) rf
   in  (length osT == length osM) && (length csT == length csM) &&
       (length vsT == length vsM) && (length rsT == length rsM) &&
       (sort osT == (sort $ map (\(Carrier  co _) -> co) osM))  &&
       (sort csT == (sort $ map (\(InterCon co _) -> co) csM))  &&
       (sort vsT == (sort $ map (\(InterVec co _) -> co) vsM))  &&
       (sort rsT == (sort $ map (\(InterRel co _) -> co) rsM))  &&
       (aritiesConsistent mo) && (namesDisjointM mo) && (namesDisjointT th) &&
       elemFormsSatisfied && vectFormsSatisfied && relaFormsSatisfied
```

Our version of game theory comes with an arbitrary homogeneous game relation on a set of positions.

```
exGameTheory = TH "GameExample" [Obj "PositionSet"] [] []
                  [Rela "B" (Obj "PositionSet") (Obj "PositionSet")] [] [] []
```

In this theory, we can formulate only whether a pair of positions admits a game transition.

```
posSet  = Obj "PositionSet"
gameRel = Rela "B" (Obj "PositionSet") (Obj "PositionSet")
posVar  = Vari "p" (Obj "PositionSet")
posVar' = Vari "q" (Obj "PositionSet")
denoteMoveInGame :: ElemTerm -> ElemTerm -> ElemForm
denoteMoveInGame v v' = PC (RCT gameRel) v v'
```

Whenever we get a concrete game relation, we extend the theory with all the constants derived from the row/column entries.

```
buildTheoryWithModelConstants :: Theory -> Model -> (Theory, Model)
buildTheoryWithModelConstants th mo =
   let MO moS osM csM vsM rsM = mo
       TH thS osT coT vsT msT eF vF rF = th
       coT' = map (\n -> Elem ("Pos " ++ [chr (48 + n)]) o') [1..k]
                 where (o',k) = head $ arities mo
       csM' = map (\n -> InterCon (Elem ("Pos " ++ [chr (48 + n)]) o') n) [1..k]
                 where (o',k) = head $ arities mo
   in  (TH thS osT coT' vsT msT eF vF rF, MO moS osM csM' vsM rsM)

isMoveInGame :: Theory -> Model -> Int -> Int -> Bool

isMoveInGame th mo n m =
   let (TH s os cs vs ms _ _ _, mo') = buildTheoryWithModelConstants th mo
       nInDomain = (1 <= n) && (n <= arity mo' (head os))
       mInDomain = (1 <= n) && (n <= arity mo' (head os))
       both   = nInDomain && mInDomain
       nConst = IndConst (Elem ("Pos " ++ [chr (48 + n)]) (head os))
       mConst = IndConst (Elem ("Pos " ++ [chr (48 + m)]) (head os))
   in  interpreteElemFormInModel mo' [] (denoteMoveInGame nConst mConst)
```

The game model first comes just with a game relation. As we ar able to add all the constants to denote row- as well as column entries, we may now ask whethter there exists a move from some position to another one.

```
gameModel = MO "GameModel"  [Carrier (Obj "PositionSet") (rows rrr)] [] []
     [InterRel (Rela "B" (Obj "PositionSet") (Obj "PositionSet")) rrr]
                 where rrr = randomMatrix 123456 20 10 10
testQuestion = isMoveInGame t m 4 3
                 where (t,m) = buildTheoryWithModelConstants exGameTheory gameModel
```

We are, however, more interested to find out in which positions a player has lost, has won, or will end up in a draw. This question cannot be answered directly from the model. This question shall serve us as an example to explain in a rather sketchy way what theory extraction is intended to mean: We execute all the computations presented earlier in connection with games and provide for means to formulate the resultsappropriately in an enhanced theory.

```
extractGameTheoryFrom th mo =
   let TH thS os cs vs ms _ _ _ = th
       numberObjects    = length os
       numberElemConsts = length cs
       numberVectConsts = length vs
       numberRelaConsts = length ms
       admissibleTheory = numberObjects    == 1   &&
                          numberElemConsts == 0   &&
                          numberVectConsts == 0   &&
                          numberRelaConsts == 1
       MO moS io ic iv ir   = mo
       numberCarriers  = length io
       numberConstants = length ic
```

```
      numberVectors   = length iv
      numberRelations = length ir
      admissibleModel = numberCarriers  == 1  &&
                        numberConstants == 0  &&
                        numberVectors   == 0  &&
                        numberRelations == 1
      (t',m') = buildTheoryWithModelConstants th mo
      TH thS' os' cs' vs' ms' eF vF rF = t'
      MO moS' io' ic' iv' ir'          = m'
      lll   = Vect "loss" (head os')
      lossT = VCT lll
      ddd   = Vect "draw" (head os')
      drawT = VCT ddd
      www   = Vect "win"  (head os')
      lodrT = lossT :||||: drawT
      winT  = NegaVect lodrT
      rT = (RCT (Rela theOnlyRelDenotation (head os) (head os)))
      theOnlyRelDenotation = (\(Rela rD _ _) -> rD) $ head ms'
      theOnlyRelation      = (\(InterRel (Rela str _ _) r) -> (str, r)) $ head ir'
      r = snd theOnlyRelation
      (aa, bb) = gameSolution r
      loss = head $ transpMat aa
      loDr = head $ transpMat bb
      draw = zipWith (\x y -> y && (not x)) loss loDr
      win  = map not loDr
  in  case admissibleTheory && admissibleModel && checkIsModelForTheory mo th &&
           (theOnlyRelDenotation == fst theOnlyRelation) of
        True  -> (TH (thS ++ "solved") os' cs'
                     [lll, ddd, www]   ms' []
                     [NegaVect lossT :<===: (rT :****: lodrT),
                      NegaVect lossT :>===: (rT :****: lodrT),
                      NegaVect lodrT :<===: (rT :****: lossT),
                      NegaVect lodrT :>===: (rT :****: lossT),
                      lossT :<===: NegaVect drawT,
                      lossT :<===: NegaVect winT,
                      drawT :<===: NegaVect winT,
                      UnivForV lossT :<===: (lossT :||||: (drawT :||||: winT))]
                                            [],
                  MO (moS ++ "solved") io' ic'
                     [InterVec lll loss, InterVec ddd draw, InterVec www win] ir')
        False -> (th, mo)       -- meaning error



exampleTest = checkIsModelForTheory m t
                where (t,m) = extractGameTheoryFrom exGameTheory gameModel
```

One will find out that the formulae $a = \overline{B{:}b}$ and $\overline{B{:}a} = b$ from our exposition on games have been entered, here however, as double inclusions. In addition, disjointness of win, draw, and loss has been formulated.

The following diagram shows the idea. There are concrete relations given and the primitive theory with which one may work on these; may be hardly more than the syntactic material. There is, however, also an ontology concerning games, irreducibility, or difunctionality, e.g. What we are constructing is

in a sense the pushout. Afterwards, the given model may be viewed with the win-loss-draw structure, e.g., which the game ontology has provided. Then the theory will also contain certain closed formulae describing what holds between the new items.

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

Theory Extraction

8 Conclusion and Outlook

With the methods presented here it is possible to analyze a given relation with regard to different concepts and to visualize the results. In modern words: We provided several ontologies in which to embed newly presented relations for better handling them in a pre-formatted way. Some programs are more efficient, others less. In particular, some cannot be really efficient as theoretical investigations show. This was not a matter of concern in this paper, though. We had in mind data of a size not too big to be handled and even visualized for presentation purposes.

There is another point of possible criticism that matters more. Should the relations handled stem from rather fuzzy sources, taken by some α-cut, e.g., the results will heavily depend on single entries of the relation. On the other hand, such a single entry may be a rather "weak" one as to its origin from the α-cut as it has hardly passed the threshold.

In the approach chosen, therefore, we have extremely high sensitivity depending on the initial data. This is a feature, one is usually not interested in. Rather, one would highly estimate results which stay more or less the same as data are changed only moderately. This would give more "meaning" to the results and would make it easier to base decisions on them.

Nonetheless, we hope that our exposition 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 the following regard:

Only recently, fuzzy relations have been investigated with more intensity. There, the entry of the matrix is not just $\mathbf{1}\,/\,\mathbf{0}$ or yes/no. Instead, coefficients from some suitable lattice are taken. In this way it is possible to express given situations in more detail. Astonishingly, much of the algebraic structure of relation algebra remains still valid. In this modified context an investigation should be executed using fuzzy relations. Reduced sensitivity of results with respect to given data may be hoped for.

9 Appendix

The present document is not just a research report but also a Haskell program in literate style. It is distributed in PDF-form as well as in ASCII-form via the internet. In order to make the ASCII-version executable, we add here some technical basics needed. The underlying ASCII document is sufficient to execute the programs in Haskell. Not every aspect of it, however, is fully explained in the present TEX-appendix.

9.1 Pretty-printing Matrices

The following functions show ways of printing matrices in ASCII-form, in TEX-form and in addition as TEX-matrices with colors

`printMat :: [[Bool]] ->` formatted output on screen
`printTeXMat :: [[Bool]] ->` formatted output string to be included in TEX-document

Sometimes we have presented boolean matrices with subdivisions so as to make some aspects more visible. To this end we used a variant of printing that in addition takes markings of row or column border lines into account that shall be marked by horizontal or vertical lines.

`printTeXMatLines :: [Bool] -> [Bool] -> [[Bool]] ->` TEX-formatted subdivided output

With `epsi 4` as the relation between a 4-element set and its powerset, the command

```
printTeXMatLines [False,True,False,False] [False,False,True,False,True,False,
     False,False,False,False,False,False,False,False,False,False] (epsi 4)
```

will produce

$$\left(\begin{array}{ccc|cc|cccccccccccc} 0&1&0&1&0&1&0&1&0&1&0&1&0&1\\ 0&0&1&1&0&0&1&1&0&0&1&1&0&0\\ \hline 0&0&0&0&1&1&1&1&0&0&0&0&1&1&1&1\\ 0&0&0&0&0&0&0&0&1&1&1&1&1&1&1&1 \end{array}\right)$$

Here, a minor problem arises since the character "\" needs special treatment as it is an escape character in Haskell. This is solved by simply using it twice in the strings.

```
stringFromPermMatTeXCol f =
   let inBoxCol x = "\\hbox to\\matrixskip{\\hfil " ++ (show x) ++ "}"
       listConvertTeXCol a  [] = a
       listConvertTeXCol [] a  = a
       listConvertTeXCol a  b  = a ++ "&\n" ++ b
   in  "\\def\\ColNames{\\hbox{\\tiny$\\smatrix{" ++
       (foldr listConvertTeXCol "" (map inBoxCol f)) ++
       "\\cr}$\\kern10pt}}"


stringFromPermMatTeXRow f =
   let inBoxRow x = "\\hbox to\\matrixskip{\\hfil " ++ (show x) ++ "}\\cr"
       listConvertTeXRow a  [] = a
       listConvertTeXRow [] a  = a
```

```
       listConvertTeXRow a  b  = a ++ "\n" ++ b
   in  "\\def\\RowNames{\\hbox{\\tiny$\\smatrix{\\noalign{\\kern-2pt}" ++
       (foldr listConvertTeXRow ""
            (map inBoxRow f)) ++ "}$}}"
```

Row and column entries as well as the matrix proper are bound together with the following function. The boolean argument `l` distinguishes whether the matrix shall be subdivided by horizontal and vertical lines, or not.

```
namedMatrix l m r c =
   let d = case l of
             True  -> druckTeXMatLinesH r c m
             False -> druckTeXMathH      m
   in  "\\def\\Matrix{\\footnotesize$" ++ d ++
       "$}\n\n\\noindent\n\n\\hbox{\\vbox{\\setbox9=\\hbox{\\" ++
       "RowNames\\Matrix}\n\\hbox to\\wd9{\\hfil\\ColNames}\n\\box9}\n}\n\n"
```

In the following function a boolean matrix can be printed together with the permuted row and column numberings.

```
stringForNamedMatrix m r c =
    (stringFromPermMatTeXCol c) ++ "\n\n" ++
    (stringFromPermMatTeXRow r) ++ "\n\n" ++
    (namedMatrix False m [] [])
```

If rows or columns are grouped according to the rearrangement with horizontal or vertical lines, we employ a different function.

```
stringForNamedMatrixLines (m, r, c, h, v) =
    (stringFromPermMatTeXCol c) ++ "\n\n" ++
    (stringFromPermMatTeXRow r) ++ "\n\n" ++
    (namedMatrix True m h v)
```

In case we print the original matrix, the unpermuted row and column numberings can be added as a default.

```
stringForOriginalNamedMatrix m =
    stringForNamedMatrix m [1..rows m] [1..cols m]
```

For an example see the relation $A_{\text{rearranged}}$ mentioned in the introduction.

9.2 Generating Random Matrices

The programming language Haskell provides for a mechanism to generate random numbers in a reproducible way. To this end one can convert any integer into a “standard generator”, which serves as a reproducible off-set.

As we are interested in random matrices with some given degree of filling density, we further provide 0 and 100 as lower and upper bound and assume a percentage parameter to be given as well as the desired row and column number.

```
randomMatrix startStdG perc r c =
  let
      rowSplit c rest = (fst cAt) : (rowSplit c (snd cAt))
                                  where cAt = splitAt c rest
      rowSplit c []   = []
      infiniteRandoms = randomRs (0 :: Int, 100 :: Int) (mkStdGen startStdG)
      alleKoeff = take (r * c) infiniteRandoms
      belowPerc = map (\ x -> fromInt x <= perc) alleKoeff
  in
      take r (rowSplit c belowPerc)
```

Here, first the randoms between 0 and 100 produced from the off-set are generated infinitely, but afterwards only $r \times c$ are actually taken. Then they are filtered according to whether they are less than or equal to the prescribed percentage. Finally, they are grouped into r rows of c elements each.

While it is easy to get a difunctional relation, e.g., it is only a rare situation to find a cyclic one with this method. So we provide better pseudo-random algorithms which anticipate the structure one is interested in to a certain extent.

A greater univalent and injective relation may be found inserting one row entry randomly and extinguishing what is in excess.

```
randomUnivAndInject startStdG m n =
   let infiniteRandoms = randomRs (1 :: Int, n :: Int) (mkStdGen startStdG)
       makeRow k  = (replicate (k-1) False) ++ [True] ++ (replicate (n-k) False)
       initialRel = map makeRow (take m infiniteRandoms)
       makeInjective r = let nonInjRows = map (\ro -> length (filter (\y -> y) ro)) r
                         in  map destroyEntries (zip nonInjRows r)
       destroyEntries (n, rr) = if n <= 1 then rr
                                          else destroyEntries (n-1, destroyFirstTrue n rr)
       destroyFirstTrue n []    = []
       destroyFirstTrue n l = if even n then destroyFirstTrue1 l
                                        else reverse (destroyFirstTrue1 (reverse l))
       destroyFirstTrue1 (h:t) = case h of
                                   True  -> False : t
                                   False -> False : (destroyFirstTrue1 t)
       univalized = transpMat (makeInjective (transpMat initialRel))
   in  makeInjective univalized
```

Here, we construct a random permutation matrix for n items.

```
randomPermutation startStdG n =
   let permutate = take n (nub (randomRs (1 :: Int, n :: Int) (mkStdGen startStdG)))
       makeRow k = (replicate (k-1) False) ++ [True] ++ (replicate (n-k) False)
   in  map makeRow permutate
```

Here, we construct an "arbitrary" irreducible and cyclic relation with n rows and block-widths between p and q.

```
randomIrredCyclic startStdG m p q =
   let startStdGG = startStdG * 421
       infiniteRandoms = randomRs (p :: Int, q :: Int) (mkStdGen startStdGG)
       groupSeq res sum (h:t) | sum + h == m = res ++ [h]
```

```
                          | sum + h <  m = groupSeq (res ++ [h]) (sum + h) t
                          | otherwise    = res ++ [m-sum]
      blockSizeSeq = groupSeq [] 0 infiniteRandoms
      blockSizeSeqShift = [last blockSizeSeq] ++ (init blockSizeSeq)
      percSeq = [25,35,45,55,65,75,85,95,100]
      matrixSeq perc = map (\(r,c) -> randomMatrix (startStdG + 27*r + 871*c) perc r c)
                       (zip blockSizeSeqShift blockSizeSeq)
      nuab a b = nulMatNM (rows a) (cols b)
      nuba a b = nulMatNM (rows b) (cols a)
      coDiagonal perc = foldl1 (\ a b -> unite [[a,        nuab a b],
                                                [nuba a b, b   ]]) (tail $ matrixSeq perc)
      nuPerc11 perc = nulMatNM (rows $ coDiagonal perc) (cols (head  $ matrixSeq perc))
      nuPerc22 perc = nulMatNM (rows (head  $ matrixSeq perc)) (cols $ coDiagonal perc)
      full perc = unite [[nuPerc11 perc,            coDiagonal perc],
                         [head  $ matrixSeq perc, nuPerc22 perc    ]]
      rP = randomPermutation (startStdG * 197) m
      iC = head (filter isIrreducible (map full percSeq))
  in  transpMat rP *** (iC *** rP)
```

9.3 Formula Translation

In order to avoid use of the clumsy syntax as long as possible, we provide for a translation of relational formulae to the element form. This means in particular to introduce all the individual variables necessary as well as quantifications which are hidden in the relational form.

```
translateVectTerm :: CatObject -> ElemVari -> VectTerm -> ElemForm

translateVectTerm o v vt =
   case vt of
      VCT vc         -> VC (VCT vc) (IndVar v)
      (rt :****: vt) -> let (o1,o2) = typeOfRelaTerm rt
                            vm = Vari  "autoVarName44" o2
                        in  ExistQuantForm vm (Conjunct (translateRelaTerm o1 o2 v vm rt)
                                                        (translateVectTerm o2 vm vt))
      vt1 :||||: vt2 -> Disjunct (translateVectTerm o v vt1) (translateVectTerm o v vt2)
      vt1 :&&&&: vt2 -> Conjunct (translateVectTerm o v vt1) (translateVectTerm o v vt2)
      NegaVect vt    -> Negated  (translateVectTerm o v vt)
      NullForV vt    -> VC (VCT (Vect "Null" o)) (IndVar v)
      UnivForV vt    -> VC (VCT (Vect "Univ" o)) (IndVar v)
      PointVector et -> let vm = (Vari  "autoVarName33" o)
                        in  Equation (IndVar vm) et

translateRelaTerm :: CatObject -> CatObject -> ElemVari -> ElemVari -> RelaTerm -> ElemForm

translateRelaTerm o o' v v' rt =
   case rt of
      RCT _          -> PC rt (IndVar v) (IndVar v')
      rt1 :***: rt2  -> let (o1,o2) = typeOfRelaTerm rt1
                            (o3,o4) = typeOfRelaTerm rt2
                            vm = Vari  "autoVarName44" o2
                        in  ExistQuantForm vm (Conjunct (translateRelaTerm o1 o2 v  vm rt1)
                                                        (translateRelaTerm o2 o4 vm v' rt2))
      rt1 :|||: rt2  -> Disjunct (translateRelaTerm o o' v v' rt1)
                                 (translateRelaTerm o o' v v' rt2)
```

```
      rt1 :&&&: rt2  -> Conjunct (translateRelaTerm o o' v v' rt1)
                                 (translateRelaTerm o o' v v' rt2)
      NegaRela  rt1  -> Negated $ translateRelaTerm o o' v v' rt1
      NullForR  rt1  -> Falsum
      UnivForR  rt1  -> Verum
      LIdent    rt1  -> Equation (IndVar v) (IndVar v')
      RIdent    rt1  -> Equation (IndVar v) (IndVar v')
      Transp    rt1  -> translateRelaTerm o' o v' v rt1
      vt1 :||--: vt2 -> Conjunct (translateVectTerm o v vt1) (translateVectTerm o' v' vt2)
      PointDiag et   -> Conjunct (Equation (IndVar v) et) (Equation (IndVar v') et)
```

```
translateVectForm :: VectForm -> ElemForm
translateVectForm (vt1 :<===: vt2) =
   let o = typeOfVectTerm vt1
       v = (Vari  "autoVarName77" o)
   in  UnivQuantForm v (Implies (translateVectTerm o v vt1) (translateVectTerm o v vt2))
translateVectForm (vt1 :>===: vt2) =
   let o = typeOfVectTerm vt1
       v = (Vari  "autoVarName88" o)
   in  UnivQuantForm v (Implies (translateVectTerm o v vt2) (translateVectTerm o v vt1))

translateRelaForm :: RelaForm -> ElemForm
translateRelaForm (rt1 :<==: rt2) =
   let (o1, o2) = typeOfRelaTerm rt1
       v1 = (Vari  "autoVarName66" o1)
       v2 = (Vari  "autoVarName55" o2)
   in  UnivQuantForm v1
         (UnivQuantForm v2 (Implies (translateRelaTerm o1 o2 v1 v2 rt1)
                                    (translateRelaTerm o1 o2 v1 v2 rt2)))
translateRelaForm (rt1 :>==: rt2) =
   let (o1, o2) = typeOfRelaTerm rt1
       v1 = (Vari  "autoVarName66" o1)
       v2 = (Vari  "autoVarName55" o2)
   in  UnivQuantForm v1
         (UnivQuantForm v2 (Implies (translateRelaTerm o1 o2 v1 v2 rt2)
                                    (translateRelaTerm o1 o2 v1 v2 rt1)))


isTotal      r = LIdent r :<==: (r :***: (Transp r))
isSurjective r = RIdent r :<==: ((Transp r) :***: r)
isUnivalent  r = (r :***: (Transp r)) :<==: LIdent r
isInjective  r = (r :***: (Transp r)) :<==: LIdent r
isMapping    r = Conjunct (translateRelaForm $ isTotal     r)
                          (translateRelaForm $ isUnivalent r)
isDifu       r = ((r :***: (Transp r)) :***: r) :<==: r

sss = PointDiag (IndVar (Vari "ddd" O1))
ttt = (RCT (Rela "e" O1 O2)) :<==: (RCT (Rela "W" O1 O2))
uuu = (RCT (Rela "e" O1 O1)) :>==: LIdent (RCT (Rela "e" O1 O2))
vvv = sss :<==: (RCT (Rela "W" O1 O1))
www = isInjective (RCT (Rela "e" O1 O2))

xXx = interpreteRelaFormInModel testModel vals (isDifu (RCT (Rela "testDifu"  O1 O2)))
```

9.4 Translation into TEX

To make the ugly type-carrying language more readable, we provide for a straightforward translation into TEX.

```
makeTeXCatObject :: CatObject -> String
makeTeXCatObject co =
   case co of
      O1    -> "O1"
      O2    -> "O2"
      O3    -> "O3"
      O4    -> "O4"
      Obj s -> s

makeTeXElemConst :: ElemConst -> String
makeTeXElemConst (Elem s o) = s

makeTeXVectConst :: VectConst -> String
makeTeXVectConst (Vect s o) = s

makeTeXRelaConst :: RelaConst -> String
makeTeXRelaConst (Rela s o o') = s

makeTeXElemVari :: ElemVari -> String
makeTeXElemVari (Vari s o) = s

makeTeXElemTerm :: ElemTerm -> String
makeTeXElemTerm et =
   case et of
      IndVar   iv -> makeTeXElemVari  iv
      IndConst ic -> makeTeXElemConst ic

makeTeXVectTerm :: VectTerm -> String
makeTeXVectTerm  vt =
   case vt of
      VCT vc         -> makeTeXVectConst vc
      rt1 :****: (vt2 :||||: vt3) -> makeTeXRelaTerm rt1 ++ "\\RELcompOP " ++
                                "(" ++ (makeTeXVectTerm (vt2 :||||: vt3)) ++ ")"
      (rt1 :|||: rt2) :****: vt3 -> "(" ++ (makeTeXRelaTerm (rt1 :|||: rt2)) ++ ")" ++
                           "\\RELcompOP " ++ (makeTeXVectTerm vt3)
      rt1 :****: (vt2 :&&&&: vt3) -> makeTeXRelaTerm rt1 ++ "\\RELcompOP " ++ "("
                                  ++ (makeTeXVectTerm (vt2 :&&&&: vt3)) ++ ")"
      (rt1 :&&&: rt2) :****: vt3 -> "(" ++ (makeTeXRelaTerm (rt1 :&&&: rt2)) ++ ")" ++
                           "\\RELcompOP " ++ (makeTeXVectTerm vt3)
      rt  :****: vt1 -> makeTeXRelaTerm rt  ++ "\\RELcompOP " ++ (makeTeXVectTerm vt1)
      vt1 :||||: vt2 -> makeTeXVectTerm vt1 ++ "\\RELorOP "   ++ (makeTeXVectTerm vt2)
      vt1 :&&&&: vt2 -> makeTeXVectTerm vt1 ++ "\\RELandOP "  ++ (makeTeXVectTerm vt2)
      NegaVect   vt1 -> "\\RELneg\\{" ++ (makeTeXVectTerm vt1) ++ "\\}"
      NullForV   vt1 -> "\\RELbot "
      UnivForV   vt1 -> "\\RELtop "
      PointVector et -> "(" ++ (makeTeXElemTerm et) ++ "\\RELcompOP\\RELtop)"

makeTeXRelaTerm :: RelaTerm -> String
makeTeXRelaTerm rt =
   case rt of
      RCT rc         -> makeTeXRelaConst rc
      rt1 :***: (rt2 :|||: rt3) -> makeTeXRelaTerm rt1 ++ "\\RELcompOP " ++ "(" ++
```

```
                                (makeTeXRelaTerm (rt2 :|||: rt3)) ++ ")"
      (rt1 :|||: rt2) :***: rt3 -> "(" ++ (makeTeXRelaTerm (rt1 :|||: rt2)) ++ ")" ++
                                "\\RELcompOP " ++
                                (makeTeXRelaTerm rt3)
      rt1 :***: (rt2 :&&&: rt3) -> makeTeXRelaTerm rt1 ++ "\\RELcompOP " ++ "(" ++
                                (makeTeXRelaTerm (rt2 :&&&: rt3)) ++ ")"
      (rt1 :&&&: rt2) :***: rt3 -> "(" ++ (makeTeXRelaTerm (rt1 :&&&: rt2)) ++ ")" ++
                                "\\RELcompOP " ++ (makeTeXRelaTerm rt3)
      rt1 :***: rt2  -> makeTeXRelaTerm rt1 ++ "\\RELcompOP " ++ (makeTeXRelaTerm rt2)
      rt1 :|||: rt2  -> makeTeXRelaTerm rt1 ++ "\\RELorOP "   ++ (makeTeXRelaTerm rt2)
      rt1 :&&&: rt2  -> makeTeXRelaTerm rt1 ++ "\\RELandOP "  ++ (makeTeXRelaTerm rt2)
      NegaRela  rt1  -> "\\RELneg\\{" ++ (makeTeXRelaTerm rt1) ++ "\\}"
      NullForR  rt1  -> "\\RELbot "
      UnivForR  rt1  -> "\\RELtop "
      LIdent    rt1  -> "\\RELide "
      RIdent    rt1  -> "\\RELide "
      Transp (rt1 :|||: rt2)  -> "(" ++ (makeTeXRelaTerm (rt1 :|||: rt2)) ++ ")\\RELtraOP "
      Transp (rt1 :&&&: rt2)  -> "(" ++ (makeTeXRelaTerm (rt1 :&&&: rt2)) ++ ")\\RELtraOP "
      Transp (rt1 :***: rt2)  -> "(" ++ (makeTeXRelaTerm (rt1 :***: rt2)) ++ ")\\RELtraOP "
      Transp    rt1  -> (makeTeXRelaTerm rt1) ++ "\\RELtraOP "
      vt1 :||--: vt2 -> makeTeXVectTerm vt1 ++ "\\RELcompOP " ++
                       (makeTeXVectTerm vt2) ++ "\\RELtraOP "
      PointDiag et   -> "(\\RELide\\RELandOP " ++ (makeTeXElemTerm et) ++
                        "\\RELcompOP\\RELtop)"

makeTeXElemForm :: ElemForm -> String
makeTeXElemForm ef =
   case ef of
      Verum               -> "{\tt True}"
      Falsum              -> "{\tt False}"
      VC    vt t          -> makeTeXElemTerm t ++ "\\in " ++ (makeTeXVectTerm vt)
      PC    rt t1 t2      -> "(" ++ (makeTeXElemTerm t1) ++ ", " ++ (makeTeXElemTerm t2)
                               ++ ")" ++ "\\in " ++ (makeTeXRelaTerm rt)
      Equation t1 t2      -> makeTeXElemTerm t1 ++ "=" ++ (makeTeXElemTerm t2)
      Negated  f1         -> "{\\tt NOT }\\;" ++ (makeTeXElemForm f1)
      Implies  f1 f2      -> makeTeXElemForm f1 ++ "\\longrightarrow " ++
                             (makeTeXElemForm f2)
      Disjunct f1 f2      -> makeTeXElemForm f1 ++ "\\;{\\tt OR}\\;" ++ (makeTeXElemForm f2)
      Conjunct f1 f2      -> makeTeXElemForm f1 ++ "\\;{\\tt AND}\\;" ++
                              (makeTeXElemForm f2)
      UnivQuantForm  v f' -> "\\forall " ++ (makeTeXElemVari v) ++ ":\\langle " ++
                                           (makeTeXElemForm f') ++ "\\rangle "
      ExistQuantForm v f' -> "\\exists " ++ (makeTeXElemVari v) ++ ":\\langle " ++
                                           (makeTeXElemForm f') ++ "\\rangle "

makeTeXVectForm :: VectForm -> String
makeTeXVectForm vf =
    case vf of
      vt1 :<===: vt2 -> makeTeXVectTerm vt1 ++ "\\RELenthOP "  ++ (makeTeXVectTerm  vt2)
      vt1 :>===: vt2 -> makeTeXVectTerm vt1 ++ "\\RELaboveOP " ++ (makeTeXVectTerm  vt2)

makeTeXRelaForm :: RelaForm -> String
makeTeXRelaForm rf =
    case rf of
      rt1 :<==: rt2 -> makeTeXRelaTerm rt1 ++ "\\RELenthOP "  ++ (makeTeXRelaTerm  rt2)
      rt1 :>==: rt2 -> makeTeXRelaTerm rt1 ++ "\\RELaboveOP " ++ (makeTeXRelaTerm  rt2)
```

Bibliography

[BGS94] Rudolf Berghammer, Thomas Gritzner, and Gunther Schmidt. Prototyping relational specifications using higher-order objects. In Jan Heering, Kurt Meinke, Bernhard Möller, and Tobias Nipkow, editors, *Higher-Order Algebra, Logic, and Term Rewriting*, volume 816 of *Lect. Notes in Comput. Sci.*, pages 56–75. Springer-Verlag, 1994. 1st Int'l Workshop, HOA '93 in Amsterdam.

[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.

[BSZ86] Rudolf Berghammer, Gunther Schmidt, and Hans Zierer. Symmetric quotients. Technical Report TUM-INFO 8620, Technische Universität München, Institut für Informatik, 1986.

[BSZ90] Rudolf Berghammer, Gunther Schmidt, and Hans Zierer. Symmetric quotients and domain constructions. *Information Processing Letters*, 33(3):163–168, 1989/90.

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

[HJW+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 `http://haskell.org/`.

[HV58] J. C. Holladay and Richard S. Varga. On powers of nonnegative matrices. *Proc. Amer. Math. Soc.*, 9:631–634, 1958.

[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.

[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.

[Var62] Richard S. Varga. *Matrix Iterative Analysis*. Prentice-Hall, Englewood Cliffs, NJ, USA, 1962.

Index