# Relational Language — Revised
# Version 2
# Draft of Oktober 25, 2004

Gunther Schmidt

Institute for Software Technology
Department of Computing Science
Federal Armed Forces University Munich, Neubiberg, Germany
e-Mail: Schmidt@informatik.unibw-muenchen.de


October 25, 2004

# Abstract

*This is the revision of the report [Sch03a]. The multilevel relational reference language has been in continuous use and development for more than one year. Several papers based on it have emerged, [OS04b, OS04a, Sch04d, Sch04a, Sch04b]. Apart from simplifications and corrections, the major new concepts are some sort of dependent types, set comprehension, pair forming, and injection in a direct sum on the element level.*

A highly expressive relational reference language is developed that covers most possibilities to use relations in practical applications, which is designed to work in a heterogeneous setting. It originated from a HASKELL-based system announced in [Sch02], forerunners of which were [HBS94, Hat97].

This language is intended to serve a variety of purposes. First, it shall allow to formulate all of the problems that have so far been tackled using relational methods with full syntax- and type-control. Transformation of relational terms and formulae in the broadest sense shall be possible as well as interpretation in many forms. In the most simple way, boolean matrices will serve as an interpretation, but also non-representable models as with the RATH-system may be used. Proofs of relational formulae in the style of RALF or in Rasiowa-Sikorski style are aimed at. Also some basics for partiality structures are included that allow to model degrees of information or degrees of availability of composite objects to be modeled.



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

Collecting several existing approaches [HBS94, Hat97, KS00, Sch02, BSW03, Sch03b], we design a relational reference language. Other activities, such as handling elementary graph theory relationally, or presenting elementary combinatorics to students, made it even more desirable to arrive at such a language. This language presented here, is intended to serve a variety of purposes.

- It shall allow to *formulate* all of the problems that have so far been tackled using relational methods, thereby offering syntax- and type-controls to reduce the likelihood of running into errors.
- It shall allow to *transform* relational terms and formulae in order to optimize these for handling them later efficiently with the help of some system. In particular, a distinction is made between the matchable denotation of an operation and its execution.
- There shall exist the possibility to *interpret* the relational language. For this mainly three ways are conceivable. In the most simple way, one shall be able to attach boolean matrices to the terms and evaluate them. In a second more sophisticated form, one shall be enabled to interpret using the RelView system. RelView is a widely known system for dealing with relations of considerable size very efficiently ([BvKU96, BBMS98, BBH+99, BH01, BHLM03, Win03]). In a third variant, interpretation shall be possible using the Rath-system. Rath is a Haskell-based system with which also nonrepresentable relation algebras may be studied.
- It is also intended to be able to *prove* relational formulae. Again, several forms shall be possible. In a first variant, a system will allow proofs in the style of Ralf, a former interactive proof assistent for executing relational proofs. Already now, however, a variant has been initiated that allows proofs in Rasiowa-Sikorski style.
- In order to support people in their work with relations, it shall be possible to *translate* relational formulae into TEX-representation or into some pure ASCII-form. Also, relational formulae shall be translated from the componentfree form into a form of first-order predicate logic.
- Finally, additional studies on partialities shall be possible. Attempts have been made to embed relation algebras into others. This means in particular to concentrate on the language used and to scrupulously distinguish which operation to apply.

All this cannot be presented in one report. Nevertheless, the matching and unification part here is meant to support designing proof systems. In a companion paper, e.g., a Rasiowa-Sikorski system will be presented. Even if it may be advisable to completely switch to one

of the well-known theorem provers, e.g., to Isabelle, the present approach may help to clarify concepts.

Not yet included is the concept of partialities, although some data type definitions and some branches with `par...` in case decompositions may be found. At the moment one should completely ignore these indications for a paper that will soon be published. It will be built as a module on top of the present one, and therefore is moderately visible already now.

We have used Haskell [HJW+92] as the programming language. It is a purely functional programming language, which is widely accepted in research and university teaching. Haskell is well-suited for logic-oriented and transformational tasks. For more information about Haskell see the Haskell WWW site at

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

The present report is written in literate style as recommended by Donald Knuth. The ASCII source text of this TEX-document is at the same time an executable Haskell program. This Haskell program is based on some modules defined separately. Together with some basic modules not mentioned in the present text, it is offered for download at

URL: http://ist.unibw-muenchen.de/Inst2/People/schmidt/RelRefLanguageHome.html

We are fully aware that many people will probably not be versed in Haskell. On the other hand, Haskell is by far the language best suited for such structural and transformational experiments. Our plan is to later care for appropriate parser elements which shall then be bound together using parser combinators to allow whatever a (reasonably precise) relation syntax desired.

The report is organized as follows. After this introduction in Ch. 1, we define the multilevel (elements, vectors, relations) language in Ch. 2 together with all the syntactic additions such as typing, well-formedness, collection of syntactic material etc. Finally, theories are introduced as Haskell data structures. Chapter 3 contains the definition of models as Haskell data structures, followed by all the functions necessary for interpretation in such a model.

As later rules and transformations shall be formulated, we introduce in Ch. 4 all the prerequisites such as substitution, matching, and unification. Along with these structural investigations, we also show how translations from the relation form to the element form may be executed, transformations into TEX or into some ASCII-form. During to the development of this report numerous tests have been executed. The main test series is included as Ch. 5. The report ends with an outlook and some acknowledgments.

# 2 A Relational Language in HASKELL

The relational language shall allow to express elements, vectors or subsets of elements, and relations in a heterogeneous setting. All syntactic means for this are collected here, including the formulation of theories. Rules to formulate transformations and the possibility of an interpretation will not yet be provided. We refer to the end of this report, where examples will illustrate the usefulness of some of the constructs now to be introduced.

## 2.1 Syntax

From the very beginning, we work in a typed or heterogeneous setting, which means that we have to provide for a language to formulate basics of a category. Normally, we will be able to give names to the category objects. For testing purposes we give some standard object names. When formulating proof rules, we will need variables for category objects. So, an automatic supply of variables with indexed names is provided for.

```
data CatObjCst = CstO String                    deriving (Eq, Show, Read, Ord)
data CatObjVar = IndexedVarO String Int |
                 VarO String                    deriving (Eq, Show, Read, Ord)
data CatObject = OC CatObjCst | OV CatObjVar | Strict ParObject |
                 DirPro CatObject CatObject  | DirSum CatObject CatObject |
                 DirPow CatObject | UnitOb | QuotMod RelaTerm |
                 InjFrom VectTerm | Lifted CatObject
                                                deriving (Eq, Show, Read, Ord)
data ParObject = ParObj CatObject | ParPro ParObject ParObject |
                 ParSum ParObject ParObject |
                 ParPow ParObject               deriving (Eq, Show, Read, Ord)
```

On the category object side also the categorical standard constructions of forming the direct product, direct sum, direct power, as well as the unit object are provided. One will recognize that forming the "quotient set" starting from a relational term (for which the property of an equivalence has to be proved first) gives a dependent type. When a vector term is given to denote a subset, we provide for the construction of the new category object injected onto it. This report does not contain further contributions to partialities; it is, however, imported as a module by some other paper based on it.

### 2.1.1 Constants and Variables

When working in first-order predicate logic, one usually needs denotations for individual variables and constants. Here, also predicate constants and predicate variables will be given and

finally relation constants and relation variables. In our setting, we always bind these together with their typing, and we restrict to unary predicates which we call vectors and binary predicates, which we call relations. Higher arities are handled via direct products.

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. [1]

```
data ElemConst = Elem String CatObject |
                 GenericElemNotation CatObject Int  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)
data FuncConst = Func String CatObject CatObject    deriving (Eq, Ord, Read, Show)
data ElemVari  = VarE String CatObject  |
                 IndexedVarE String Int CatObject deriving (Eq, Ord, Read, Show)
data VectVari  = VarV String CatObject |
                 IndexedVarV String Int CatObject deriving (Eq, Ord, Read, Show)
data RelaVari  = VarR String CatObject CatObject |
                 IndexedVarR String Int CatObject
                                      CatObject deriving (Eq, Ord, Read, Show)
```

The function constant is not really necessary as we have relation constants. In many cases, however, questions at the borderline between first-order theory and relation theory are to be handled where it is advisable to have them.

An automatic variable index supply is organized, i.e., it will be possible to generate new element variables automatically using the following function.

```
nextIndexE :: [ElemVari] -> Int
nextIndexE l = let isIndexedVarE o  = case o of IndexedVarE _ _ _  -> True
                                                _                  -> False
                   indexInVar    ev = case ev of IndexedVarE s i co -> i
                   indicesUsed = map indexInVar (filter isIndexedVarE l)
               in  1 + maximum (0 : indicesUsed)

nextElemVari :: CatObject -> [ElemVari] -> ElemVari
nextElemVari o evs = let j = nextIndexE evs
                     in  IndexedVarE "e" j o

nextIndexV :: [VectVari] -> Int
nextIndexV l = let isIndexedVarV o  = case o of IndexedVarV _ _ _  -> True
                                                _                  -> False
                   indexInVar vv = case vv of IndexedVarV s i co -> i
                   indicesUsed = map indexInVar (filter isIndexedVarV l)
               in  1 + maximum (0 : indicesUsed)

nextVectVari :: CatObject -> [VectVari] -> VectVari
nextVectVari o vvs = let j = nextIndexV vvs
```

[1] The generic element notation is a hidden one. It is only used when running through a set of elements over a base set prior to its interpretation.

```
                    in  IndexedVarV "v" j o

nextIndexR :: [RelaVari] -> Int
nextIndexR l = let isIndexedVarR o  = case o of IndexedVarR _ _ _ _  -> True
                                                _                    -> False
                   indexInVar rv = case rv of IndexedVarR s i co co' -> i
                   indicesUsed = map indexInVar (filter isIndexedVarR l)
               in  1 + maximum (0 : indicesUsed)

nextRelaVari :: CatObject -> CatObject -> [RelaVari] -> RelaVari
nextRelaVari o o' rvs = let j = nextIndexR rvs
                        in  IndexedVarR "R" j o o'

elemVarName v = case v of VarE        string   _ -> string
                          IndexedVarE string i _ -> string ++ show i
```

With the following function, one may generate a number of object, element, vector, relation, and function variables as required.

```
supply :: Int -> Int -> Int -> Int -> Int -> Int ->
   ([CatObjVar],[ElemVari],[VectVari],[RelaVari],[FormVari])
supply i o e v r f =
   let indices0 = [(i+1) ..]
       (oIndices,indices1) = splitAt o indices0
       (eObjIndi,indices2) = splitAt e indices1
       (vObjIndi,indices3) = splitAt v indices2
       (rDOMIndi,indices4) = splitAt r indices3
       (rCODIndi,indices5) = splitAt r indices4
       (eIndices,indices6) = splitAt e indices5
       (vIndices,indices7) = splitAt v indices6
       (rIndices,indices8) = splitAt r indices7
       (fIndices,indices9) = splitAt f indices8
       oVars  = map (IndexedVarO "o") oIndices
       eVars  = zipWith  (\x y   -> IndexedVarE "e" x (OV $ IndexedVarO "o" y))
                         eIndices eObjIndi
       vVars  = zipWith  (\x y   -> IndexedVarV "v" x (OV $ IndexedVarO "o" y))
                         vIndices vObjIndi
       rVars  = zipWith3 (\x y z -> IndexedVarR "R" x (OV $ IndexedVarO "o" y)
                                                      (OV $ IndexedVarO "o" z))
                         rIndices rDOMIndi rCODIndi
       fVars = map (IndexedVarF "f") fIndices
   in  (oVars,eVars,vVars,rVars,fVars)
```

Sometimes, one needs several object, element, vector, relation, and function constants.

```
supplyConst :: [String] -> [(String,String)] -> [(String,String)] ->
   [(String,String,String)] -> [(String,String,String)] ->
          ([CatObjCst],[ElemConst],[VectConst],[RelaConst],[FuncConst])
supplyConst o e v r f =
   let oConsts  = map CstO o
       oTerms   = map OC   oConsts
```

```
      oWithName n = head $ filter (\(OC (CstO n')) -> n == n') oTerms
      eConsts  = map (\(x,y)   -> Elem x (oWithName y)) e
      vConsts  = map (\(x,y)   -> Vect x (oWithName y)) v
      rConsts  = map (\(x,y,z) -> Rela x (oWithName y) (oWithName z)) r
      fConsts  = map (\(x,y,z) -> Func x (oWithName y) (oWithName z)) f
  in  (oConsts,eConsts,vConsts,rConsts,fConsts)
```

With the latter construction constants may be generated with prescribed names. One has to provide the arguments as

```
[desiredCatObjConstName]
[(desiredElemConstName, itsDesiredDomainName)]
[(desiredVectConstName, itsDesiredDomainName)]
[(desiredRelaConstName, itsDesiredDomainName, itsDesiredRangeName)]
[(desiredFuncConstName, itsDesiredDomainName, itsDesiredRangeName)]
```

### 2.1.2 Terms

On all this, we now build first-order predicate logic, introducing 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.

The element terms constructed via `Some, That` need special explanation. They are correctly defined only if, e.g., the `vt` in `ThatV vt` denotes a point. In `SomeR rt`, the `rt` must denote a nonempty part of the identity. Later, typically a proof obligation will be issued to guarantee such properties.

```
data ElemTerm = EV ElemVari | EC ElemConst | Pair ElemTerm ElemTerm |
                Inj1 ElemTerm CatObject | Inj2 CatObject ElemTerm |
                ThatV VectTerm | SomeV VectTerm | ThatR RelaTerm  | SomeR RelaTerm |
                FuncAppl FuncConst ElemTerm | VectToElem VectTerm |
                EFctAppl ElemFct   ElemTerm           deriving (Eq, Ord, Read, Show)
data VectTerm = VC VectConst | VV VectVari | RelaTerm :****: VectTerm |
                VectTerm :||||: VectTerm | VectTerm :&&&&: VectTerm |
                NegaV VectTerm | NullV CatObject | UnivV CatObject |
                SupVect VectSET | InfVect VectSET | PointVect ElemTerm |
                Syq RelaTerm VectTerm | RelaToVect RelaTerm |
                PowElemToVect ElemTerm |
                VFctAppl VectFct VectTerm             deriving (Eq, Ord, Read, Show)
data RelaTerm = RC RelaConst | RV RelaVari | RelaTerm :***: RelaTerm |
                RelaTerm :|||: RelaTerm | RelaTerm :&&&: RelaTerm |
                NegaR RelaTerm | Ident CatObject | NullR CatObject CatObject |
                UnivR CatObject CatObject | Convs RelaTerm |
                VectTerm :||--: VectTerm | SupRela RelaSET | InfRela RelaSET |
                RelaTerm :*: RelaTerm | RelaTerm :\/: RelaTerm |
                Pi   CatObject CatObject | Rho   CatObject CatObject |
                Iota CatObject CatObject | Kappa CatObject CatObject |
                CASE RelaTerm RelaTerm | Project RelaTerm |
                Epsi CatObject | PointDiag ElemTerm | SyQ RelaTerm RelaTerm |
                Wait PartTerm | Belly PartTerm | InjTerm VectTerm |
                ProdVectToRela VectTerm | PartOrd ParObject |
```

```
                RFctAppl RelaFct ArgEVR                deriving (Eq, Ord, Read, Show)
```

A column vector multiplied via `:||--:` with a row vector will deliver a relation. The construct `RelaTerm :****: VectTerm` is intended to model the Peirce product. With `Pi, Rho` generic denotations for projections from a direct product are introduced; with `Iota, Kappa` generic denotations for the injections into a direct sum. Finally, `Epsi` generically denotes the relationship between a set and its powerset. It is suggested that the generic constructs always be introduced and named consistently when needed.

```
introduceNamesForSixProdConstituents s t =
   let p    = DirPro s t
       pi   = Pi  s t
       rho  = Rho s t
       piT  = Convs pi
       rhoT = Convs rho
       strictFork a  b  = (a :***: piT) :&&&: (b :***: rhoT)
   in  (p,pi,piT,rho,rhoT,strictFork)

introduceNamesForSixSumConstituents s t =
   let s   = DirSum s t
       io  = Iota  s t
       ka  = Kappa s t
       ioT = Convs io
       kaT = Convs ka
       caseDist a b = (ioT :***: a) :|||: (kaT :***: b)
   in  (s,io,ioT,ka,kaT,caseDist)

introduceNamesForThreePowerConstituents s =
   let po    = DirPow s
       epsi  = Epsi   s
       epsiT = Convs epsi
   in  (po,epsi,epsiT)
introduceNamesForDependentSubsetAndInjection vt =
   let subsetType = InjFrom vt
       injection  = InjTerm vt
   in  (subsetType,injection)
introduceNamesForDependentQuotientAndProjection rt =
   let quotientType = QuotMod rt
       projection   = Project rt
   in  (quotientType,projection)
```

The partiality terms are only included because of their cross-relationship with the relational terms, but not yet discussed here.

```
data PartTerm = Lift RelaTerm | PartTerm :*****: PartTerm |
                Fetus ParObject RelaTerm ParObject |
                PartTerm :|||||: PartTerm | PartTerm :&&&&&: PartTerm |
                NegaPart PartTerm | IdentP ParObject | NullP ParObject ParObject |
                UnivP ParObject ParObject | TranspP PartTerm |
                PPi ParObject ParObject | PRho ParObject ParObject |
```

```
                PartTerm :#: PartTerm | PartTerm :\\//: PartTerm |
                PIota ParObject ParObject | PKappa ParObject ParObject |
                PCASE PartTerm PartTerm | PEpsi ParObject  |
                LiftL RelaTerm | LiftR RelaTerm |           --only temporarily
                Noc ParObject ParObject | WaitL RelaTerm |  --
                WaitR RelaTerm | ProjectL RelaTerm |        --for use in 4.4
                ProjectR RelaTerm | PHI RelaTerm RelaTerm
                                                  deriving (Eq, Ord, Read, Show)
```

The following are necessary when, e.g., introducing a transitive closure of a relation by the classical infimum definition. We must be able to write down a functional. As we follow an overly strict typing discipline for reasons of precision, we have to formally characterize the variables and the argument terms. So far, pairs of *relations*, e.g., will not be accepted as arguments, while tuples of *elements* may be introduced via direct products.

```
data EVRVari  = EVar ElemVari | VVar VectVari | RVar RelaVari
                                                  deriving (Eq, Ord, Read, Show)
data ArgEVR   = ArgE ElemTerm | ArgV VectTerm | ArgR RelaTerm
                                                  deriving (Eq, Ord, Read, Show)
data ElemFct  = EFCT ElemVari ElemTerm            deriving (Eq, Ord, Read, Show)
data VectFct  = VFCT VectVari VectTerm            deriving (Eq, Ord, Read, Show)
data RelaFct  = RFCT EVRVari  RelaTerm            deriving (Eq, Ord, Read, Show)
```

### 2.1.3 Sets of Elements, Vectors, and Relations

In order to be able to formulate formulae on least upper bounds, e.g., also sets shall be formed. They are either sets given by some condition or explicit sets, again distinguished for elements, vectors, and relations. For the explicit sets also the type is provided, a measure which is only important in case of void sets. In the case of elements, we may run over all elements of the category object.

```
data ElemSET = VarES String CatObject | ET CatObject |
               ES ElemVari [Formula]  | EX [ElemTerm] CatObject
                                             deriving (Eq, Show, Read, Ord)
data VectSET = VarVS String CatObject |
               VS VectVari [Formula]  | VX [VectTerm] CatObject
                                             deriving (Eq, Show, Read, Ord)
data RelaSET = VarRS String CatObject CatObject | RT RelaFct ElemSET |
               RS RelaVari [Formula] | RX [RelaTerm] CatObject CatObject
                                             deriving (Eq, Show, Read, Ord)
```

### 2.1.4 Formulae

Three sorts of formulae are distinguished in order to maintain typing as long as possible. Only when negation, e.g., is applied to a formula, it will be handled as a formula. Until that point, the type is a convenient way of correctness control. For the time being, `:<=/=:` is a second way of negating in a special context, that allows easier pattern matching.

```
data UnivOrExist = Univ | Exis                          deriving (Eq, Show, Read, Ord)
switchUE :: UnivOrExist -> UnivOrExist
switchUE ue = case ue of Univ -> Exis
                         Exis -> Univ

data CatOForm = ObjEqual CatObject CatObject            deriving (Eq, Ord, Read, Show)
data ElemForm = Equation ElemTerm  ElemTerm | NegaEqua ElemTerm  ElemTerm |
                QuantElemForm UnivOrExist ElemVari [Formula]
                                                        deriving (Eq, Ord, Read, Show)
data VectForm = VectTerm :<===: VectTerm | VectTerm :>===: VectTerm |
                VectTerm :====: VectTerm | VectTerm :<=/=: VectTerm |
                VectTerm :==/=: VectTerm |
                VectTerm :>=/=: VectTerm | VE VectTerm ElemTerm |
                VectInSet VectTerm VectSET |
                QuantVectForm UnivOrExist VectVari [Formula]
                                                        deriving (Eq, Ord, Read, Show)
data RelaForm = RelaTerm :<==: RelaTerm | RelaTerm :>==: RelaTerm |
                RelaTerm :===: RelaTerm | RelaTerm :<=/: RelaTerm |
                RelaTerm :=/=: RelaTerm |
                RelaTerm :>=/: RelaTerm | RelaInSet RelaTerm RelaSET |
                REE RelaTerm ElemTerm ElemTerm |
                QuantRelaForm UnivOrExist RelaVari [Formula]
                                                        deriving (Eq, Ord, Read, Show)


data PartForm = PartTerm :<====: PartTerm | PartTerm :>====: PartTerm |
                PartTerm :<=/==: PartTerm | PartTerm :>=/==: PartTerm |
                StrictPartContained PartTerm PartTerm
                --PartInSet PartTerm PartSET | REE PartTerm ElemTerm ElemTerm |
                --UnivQuantPartForm  PartVari [Formula] |
                --ExistQuantPartForm PartVari [Formula]
                                                        deriving (Eq, Ord, Read, Show)

data FormVari = VarF String | IndexedVarF String Int deriving (Eq, Ord, Read, Show)
data Formula  = FV FormVari | OF CatOForm | EF ElemForm |
                VF VectForm | RF RelaForm | PF PartForm |
                Verum | Falsum | Negated Formula | Implies Formula Formula |
                SemEqu   Formula Formula |
                Disjunct Formula Formula | Conjunct Formula Formula
                                                        deriving (Eq, Ord, Read, Show)
```

The type of a formula is in all five cases intended to be `Bool`; we have, however, tried to benefit from typing of vectors and relations as long as possible.

## 2.2 Expansions

The operations `:*:` and `:\/:` are defined using the other operations. As we have included them here, one may later match them. Here are their expanded versions.

```
expandDefinedElemTerm det =
   case det of VectToElem vt -> ThatV $ Syq (Epsi $ domVT vt) vt
               _             -> error "situation for VectToElem must not occur"

expandDefinedVectTerm dvt =
   case dvt of RelaToVect rt -> let (d,c) = typeOfRT rt
                                    pd = DirPro d c
                                    ppppi = Pi  d c
                                    rrrho = Rho d c
                                in  (ppppi :***: rt :&&&: rrrho) :****: UnivV c
               Syq rt vt  -> let rtT  = Convs rt
                                 rtTN = NegaR rtT
                                 vtN  = NegaV vt
                                 rtTvtNQuer = NegaV (rtT  :****: vtN)
                                 rtTNvtQuer = NegaV (rtTN :****: vt)
                             in  rtTvtNQuer :&&&&: rtTNvtQuer
               PowElemToVect et ->
                  case domET et of
                     DirPow o -> Epsi o :****: (PointVect et)
                     _        -> error "Domain must be direct power"
               _          -> dvt

expandDefinedRelaTerm drt =
   case drt of
      rt1 :*: rt2 ->  let d1    = domRT rt1
                          d2    = domRT rt2
                          c1    = codRT rt1
                          c2    = codRT rt2
                          pi1   = Pi d1 d2
                          pi2   = Pi c1 c2
                          rho1  = Rho d1 d2
                          rho2  = Rho c1 c2
                          pi2T  = Convs pi2
                          rho2T = Convs rho2
                          firstPart = pi1 :***: (rt1 :***: pi2T)
                          secndPart = rho1 :***: (rt2 :***: rho2T)
                      in  firstPart :&&&: secndPart
      rt1 :\/: rt2 -> let c1    = codRT rt1
                          c2    = codRT rt2
                          pi2   = Pi c1 c2
                          rho2  = Rho c1 c2
                          pi2T  = Convs pi2
                          rho2T = Convs rho2
                          firstPart = rt1 :***: pi2T
                          secndPart = rt2 :***: rho2T
                      in  firstPart :&&&: secndPart
      SyQ rt1 rt2  -> let rt1T  = Convs rt1
                          rt1TN = NegaR rt1T
                          rt2N  = NegaR rt2
                          rt1Trt2NQuer = NegaR (rt1T  :***: rt2N)
```

```
                        rt1TNrt2Quer = NegaR (rt1TN :***: rt2)
                    in  rt1Trt2NQuer :&&&: rt1TNrt2Quer
      ProdVectToRela vt ->
         case domVT vt of
            DirPro o o' -> let pppi = Pi  o o'
                               rrho = Rho o o'
                           in  Convs pppi :***: (rrho :&&&: (vt :||--: (UnivV o')))
            _           -> error "Domain must be direct product"
      _             -> drt

expandDefinedPartTerm dt =
   case dt of
      rt1 :#: rt2 ->    let d1    = domPT rt1
                            d2    = domPT rt2
                            c1    = codPT rt1
                            c2    = codPT rt2
                            pi1   = PPi d1 d2
                            pi2   = PPi c1 c2
                            rho1  = PRho d1 d2
                            rho2  = PRho c1 c2
                            pi2T  = TranspP pi2
                            rho2T = TranspP rho2
                            firstPart = pi1 :*****: (rt1 :*****: pi2T)
                            secndPart = rho1 :*****: (rt2 :*****: rho2T)
                        in  firstPart :&&&&&: secndPart
      rt1 :\\//: rt2 -> let d     = domPT rt1
                            c1    = codPT rt1
                            c2    = codPT rt2
                            pi2   = PPi c1 c2
                            rho2  = PRho c1 c2
                            pi2T  = TranspP pi2
                            rho2T = TranspP rho2
                            firstPart = rt1 :*****: pi2T
                            secndPart = rt2 :*****: rho2T
                        in  firstPart :&&&&&: secndPart


expandDefinedVectForm dvf =
   case dvf of
      vt1 :>===: vt2 ->           VF (vt2 :<===: vt1)
      vt1 :====: vt2 -> Conjunct (VF (vt1 :<===: vt2))
                                 (VF (vt2 :<===: vt1))
      vt1 :<=/=: vt2 -> Negated  (VF (vt1 :<===: vt2))
      vt1 :>=/=: vt2 -> Negated  (VF (vt2 :<===: vt1))
      vt1 :==/=: vt2 -> Disjunct (VF (vt1 :<=/=: vt2))
                                 (VF (vt2 :<=/=: vt1))
expandDefinedRelaForm drf =
   case drf of
      rt1 :>==: rt2 -> RF (rt2 :<==: rt1)
      rt1 :===: rt2 -> Conjunct (RF (rt1 :<==: rt2))
```

```
                                (RF (rt2 :<==: rt1))
      rt1 :<=/: rt2 -> Negated  (RF (rt1 :<==: rt2))
      rt1 :>=/: rt2 -> Negated  (RF (rt2 :<==: rt1))
      rt1 :=/=: rt2 -> Disjunct (RF (rt1 :<=/: rt2))
                                (RF (rt2 :<=/: rt1))
expandDefinedFormula df =
   case df of
      SemEqu f1 f2 -> Conjunct (Implies f1 f2)
                               (Implies f2 f1)


isNegatedRelaTerm (rt,_) = case rt of NegaR _ -> True
                                      _       -> False
```

## 2.3 Standard Properties

There are lots of properties one will use at every moment, mainly concerning function and orderig properties. We provide here for formulations in the proposed language.

```
isReflexiveFormula rt =
   Conjunct (RF $ Ident (domRT rt) :<==: rt)
            (OF $ ObjEqual (domRT rt) (codRT rt))
isIrreflexiveFormula rt =
   Conjunct (RF $ Ident (domRT rt) :<==: (NegaR rt))
            (OF $ ObjEqual (domRT rt) (codRT rt))
isSymmetricFormula rt =
   Conjunct (RF $ Convs rt :<==: rt)
            (OF $ ObjEqual (domRT rt) (codRT rt))
isAsymmetricFormula rt =
   Conjunct (RF $ Convs rt :<==: (NegaR rt))
            (OF $ ObjEqual (domRT rt) (codRT rt))
isAntisymmetricFormula rt =
   Conjunct (RF $ Convs rt :&&&: rt :<==: (Ident (domRT rt)))
            (OF $ ObjEqual (domRT rt) (codRT rt))
isTransitiveFormula rt =
   Conjunct (RF $ rt :***: rt :<==: rt)
            (OF $ ObjEqual (domRT rt) (codRT rt))
isPreorderFormula rt =
   Conjunct (isReflexiveFormula rt) (isTransitiveFormula rt)
isOrderFormula rt =
   Conjunct (isPreorderFormula rt) (isAntisymmetricFormula rt)
isStrictorderFormula rt =
   Conjunct (isTransitiveFormula rt) (isIrreflexiveFormula rt)
isUnivalentFormula rt =
   RF $ Convs rt :***: rt :<==: (Ident (codRT rt))
isInjectiveFormula rt =
   isUnivalentFormula (Convs rt)
isTotalFormula rt =
   RF $ Ident (domRT rt) :<==: (rt :***: (Convs rt))
```

```
isSurjectiveFormula rt =
   isTotalFormula (Convs rt)
isMappingFormula rt =
   Conjunct (isTotalFormula rt) (isUnivalentFormula rt)
isPermutatioinFormula rt =
   Conjunct (isMappingFormula rt) (isMappingFormula (Convs rt))
isDifunctionalFormula rt =
   RF $ rt :***: (Convs rt) :***: rt :<==: rt
isRowconstantFormula rt =
   RF $ UnivR (domRT rt) UnitOb :<==: (rt :***: (UnivR (codRT rt) UnitOb))
isColconstantFormula rt =
   isRowconstantFormula (Convs rt)
```

## 2.4 Syntactic Standard Recursions

Over these definitions the usual recursive algorithms are defined. They will determine domain and range of a term, collect the syntactical material it is built from, collect the free or bound variables, they will qualify a term to be well-formed, etc.

### 2.4.1 Inductive Typing Definitions

From the very beginning, we may ask for the arities of these constants and variables.

```
domEC :: ElemConst -> CatObject
domEC ec = case ec of Elem                _ o -> o
                      GenericElemNotation o _ -> o
domEV :: ElemVari  -> CatObject
domEV ev = case ev of VarE          _   o -> o
                      IndexedVarE _ _ o -> o

domVC :: VectConst -> CatObject
domVC (Vect _ o)   = o
domVV :: VectVari  -> CatObject
domVV vv = case vv of VarV          _   o -> o
                      IndexedVarV _ i o -> o

domRC, codRC :: RelaConst -> CatObject
domRC (Rela _ o _) = o
codRC (Rela _ _ o) = o
domRV, codRV :: RelaVari -> CatObject
domRV rv = case rv of VarR          _   o _ -> o
                      IndexedVarR _ i o _ -> o
codRV rv = case rv of VarR          _   _ o -> o
                      IndexedVarR _ i _ o -> o
domFC, codFC :: FuncConst -> CatObject
domFC (Func _ o _) = o
codFC (Func _ _ o) = o
```

```
domET :: ElemTerm -> CatObject
domET et =
   case et of
      EV   ev      -> domEV ev
      EC   ec      -> domEC ec
      Pair et1 et2 -> DirPro (domET et1) (domET et2)
      Inj1 et o    -> DirSum (domET et) o
      Inj2 o  et   -> DirSum          o (domET et)
      ThatV vt -> domVT vt
      SomeV vt -> domVT vt
      ThatR rt -> domRT rt
      SomeR rt -> domRT rt
      FuncAppl _ et -> domET et
      VectToElem _ -> domET $ expandDefinedElemTerm et


domVT :: VectTerm -> CatObject
domVT vt1 =
   case vt1 of
      VC          vc   -> domVC vc
      VV          vv   -> domVV vv
      rt :****:   vt   -> domRT rt
      vt :||||:   vt'  -> domVT vt
      vt :&&&&:   vt'  -> domVT vt
      Syq      rt vt'  -> codRT rt
      NegaV       vt   -> domVT vt
      NullV       o    -> o
      UnivV       o    -> o
      SupVect     vs   -> domVS vs
      InfVect     vs   -> domVS vs
      RelaToVect  _    -> domVT $ expandDefinedVectTerm vt1
      VFctAppl vf vt   -> domFV vf
      PointVect   et   -> domET et
      PowElemToVect et -> domVT $ expandDefinedVectTerm vt1


domRT, codRT :: RelaTerm -> CatObject
domRT rt1 =
   case rt1 of
      RC        rc  -> domRC rc
      RV        rv  -> domRV rv
      rt :***:  rt' -> domRT rt
      rt :|||:  rt' -> domRT rt
      rt :&&&:  rt' -> domRT rt
      NegaR     rt  -> domRT rt
      Ident     o   -> o
      NullR     o _ -> o
      UnivR     o _ -> o
      Convs     rt  -> codRT rt
      vt :||--: vt' -> domVT vt
      SupRela   rs  -> domRS rs
```

```
      InfRela   rs  -> domRS rs
      (:*:)   _ _   -> domRT $ expandDefinedRelaTerm rt1
      (:\/:)  _ _   -> domRT $ expandDefinedRelaTerm rt1
      Pi      o o'  -> DirPro o o'
      Rho     o o'  -> DirPro o o'
      Iota    o _   -> o
      Kappa   _ o'  -> o'
      CASE rt1 rt2  -> DirSum (domRT rt1) (domRT rt2)
      Project  rt2  -> domRT rt2
      Epsi    o     -> o
      PointDiag et  -> domET et
      SyQ     _ _   -> codRT rt1
      --Wait      rt  -> case domPT rt of
      Belly pt  -> domRT $ PartOrd $ domPT pt
      --                    ParObj o -> o
      --                    x        -> Strict x
      InjTerm   vt  -> InjFrom vt
      ProdVectToRela _ -> domRT $ expandDefinedRelaTerm rt1
      PartOrd po       ->
         case po of
            ParObj co1     -> Lifted co1
            ParPro po1 po2 -> DirPro (domRT $ PartOrd po1) (domRT $ PartOrd po2)
            ParSum po1 po2 -> DirSum (domRT $ PartOrd po1) (domRT $ PartOrd po2)
            ParPow po1     -> DirPow (domRT $ PartOrd po1)
      RFctAppl rf rt   -> domFR rf

codRT rt1 =
   case rt1 of
      RC        rc  -> codRC rc
      RV        rv  -> codRV rv
      rt :***:  rt' -> codRT rt'
      rt :|||:  rt' -> codRT rt
      rt :&&&:  rt' -> codRT rt
      NegaR     rt  -> codRT rt
      Ident     o   -> o
      NullR     _ o -> o
      UnivR     _ o -> o
      Convs     rt  -> domRT rt
      vt :||--: vt' -> domVT vt'
      SupRela   rs  -> codRS rs
      InfRela   rs  -> codRS rs
      (:*:)   _ _   -> codRT $ expandDefinedRelaTerm rt1
      (:\/:)  _ _   -> codRT $ expandDefinedRelaTerm rt1
      Pi      o _   -> o
      Rho     _ o'  -> o'
      Iota    o o'  -> DirSum o o'
      Kappa   o o'  -> DirSum o o'
      CASE rt1 _    -> codRT rt1
      Project   rt  -> QuotMod rt
      Epsi    o     -> DirPow o
```

```
      PointDiag et  -> domET et
      SyQ   _ rt2   -> codRT rt2
      --Wait      rt  -> case codPT rt of
      Belly pt -> codRT $ PartOrd $ codPT pt
      --                    ParObj o -> o
      --                    x        -> Strict x
      InjTerm   vt  -> domVT vt
      ProdVectToRela _ -> codRT $ expandDefinedRelaTerm rt1
      PartOrd po       ->
         case po of
            ParObj co1     -> Lifted co1
            ParPro po1 po2 -> DirPro (codRT $ PartOrd po1) (codRT $ PartOrd po2)
            ParSum po1 po2 -> DirSum (codRT $ PartOrd po1) (codRT $ PartOrd po2)
            ParPow po1     -> DirPow (codRT $ PartOrd po1)
      RFctAppl rf rt -> codFR rf

domPT, codPT :: PartTerm -> ParObject
domPT pt =
   case pt of
      Lift rt       -> case domRT rt of
                           Strict x -> x
                           _        -> ParObj (domRT rt)
      Fetus po1 _ _ -> po1
      pt1 :*****: _ -> domPT pt1
      pt1 :|||||: _ -> domPT pt1
      pt1 :&&&&&: _ -> domPT pt1
      NegaPart pt1  -> domPT pt1
      IdentP po     -> po
      NullP  d _    -> d
      UnivP  d _    -> d
      TranspP pt1   -> codPT pt1
      PPi    o o'   -> ParPro o o'
      PRho   o o'   -> ParPro o o'
      pt1 :#: pt2   -> ParPro (domPT pt1) (domPT pt2)
      pt1 :\\//: _  -> domPT pt1
      PIota  o o'   -> o
      PKappa o o'   -> o'
      PCASE pt1 pt2 -> ParSum (domPT pt1) (domPT pt2)
      PEpsi  o      -> o
codPT pt =
   case pt of
      Lift rt        -> case codRT rt of
                              Strict x -> x
                              _        -> ParObj (codRT rt)
      Fetus _ _ po2  -> po2
      _ :*****: pt2  -> codPT pt2
      _ :|||||: pt2  -> codPT pt2
      _ :&&&&&: pt2  -> codPT pt2
      NegaPart pt1   -> codPT pt1
      IdentP po      -> po
```

```
      NullP  _ d     -> d
      UnivP  _ d     -> d
      TranspP pt1    -> domPT pt1
      PPi    o _     -> o
      PRho   _ o'    -> o'
      pt1 :#: pt2    -> ParPro (codPT pt1) (codPT pt2)
      pt1 :\\//: pt2 -> ParPro (codPT pt1) (codPT pt2)
      PIota  o o'    -> ParSum o o'
      PKappa o o'    -> ParSum o o'
      PCASE pt1 _    -> codPT pt1
      PEpsi  o       -> ParPow o


domFE :: ElemFct -> CatObject
domFE (EFCT ev et) = domET et
domFV :: VectFct -> CatObject
domFV (VFCT vv vt) = domVT vt
domFR, codFR :: RelaFct -> CatObject
domFR (RFCT _ rt) = domRT rt
codFR (RFCT _ rt) = codRT rt

domES es =
   case es of
      VarES _  o -> o
      ES ev _ -> domEV ev
      ET    o -> o
      EX _  o -> o
domVS vs =
   case vs of
      VarVS _  o -> o
      VS    vv _ -> domVV vv
      VX    _  o -> o
domRS rs =
   case rs of
      VarRS _ o o' -> o
      RS rv _      -> domRV rv
      RT f  _      -> domFR f
      RX _  o _    -> o
codRS rs =
   case rs of
      VarRS _ o o' -> o'
      RS rv _      -> codRV rv
      RT f  _      -> codFR f
      RX _  _ o'   -> o'

domEF :: ElemForm -> CatObject
domEF ef =
   case ef of
      Equation        et1 _ -> domET et1
      NegaEqua        et1 _ -> domET et1
      QuantElemForm _ ev  _ -> domEV ev
```

```
domVF :: VectForm -> CatObject
domVF vf =
   case vf of
      vt1 :<===: _ -> domVT vt1
      vt1 :>===: _ -> domVT vt1
      vt1 :====: _ -> domVT vt1
      vt1 :<=/=: _ -> domVT vt1
      vt1 :>=/=: _ -> domVT vt1
      vt1 :==/=: _ -> domVT vt1
      VE        vt _ -> domVT vt
      VectInSet vt _ -> domVT vt
      QuantVectForm _ vv _ -> domVV vv

domRF :: RelaForm -> CatObject
domRF rf =
   case rf of
      rt1 :<==: _ -> domRT rt1
      rt1 :>==: _ -> domRT rt1
      rt1 :===: _ -> domRT rt1
      rt1 :<=/: _ -> domRT rt1
      rt1 :>=/: _ -> domRT rt1
      rt1 :=/=: _ -> domRT rt1
      REE       rt _ _ -> domRT rt
      RelaInSet rt _   -> domRT rt
      QuantRelaForm _ rv _ -> domRV rv
codRF :: RelaForm -> CatObject
codRF rf =
   case rf of
      rt1 :<==: _ -> codRT rt1
      rt1 :>==: _ -> codRT rt1
      rt1 :===: _ -> codRT rt1
      rt1 :<=/: _ -> codRT rt1
      rt1 :>=/: _ -> codRT rt1
      rt1 :=/=: _ -> codRT rt1
      REE       rt _ _ -> codRT rt
      RelaInSet rt _   -> codRT rt
      QuantRelaForm _ rv _ -> codRV rv
domPF :: PartForm -> ParObject
domPF rf =
   case rf of
      rt1 :<====: rt2   -> domPT rt1
      rt1 :>====: rt2   -> domPT rt1
      rt1 :<=/==: rt2   -> domPT rt1
      rt1 :>=/==: rt2   -> domPT rt1
      StrictPartContained pt1 pt2 -> domPT pt1
codPF :: PartForm -> ParObject
codPF rf =
   case rf of
      rt1 :<====: rt2   -> codPT rt1
```

```
      rt1 :>====: rt2   -> codPT rt1
      rt1 :<=/==: rt2   -> codPT rt1
      rt1 :>=/==: rt2   -> codPT rt1
      StrictPartContained pt1 pt2 -> codPT pt1
```

Collecting this in a type class definition, one may henceforth simply write `dom, cod`. For convenience, we allow already now to ask for type and for well-formedness.

```
class Typed a where
   dom :: a -> CatObject
   cod :: a -> CatObject
   isWellFormed :: a -> Bool
   typeOf :: a -> (CatObject,CatObject)
   syntMat :: a -> ([CatObjVar],[CatObjCst],[ElemVari],[ElemConst],[VectVari],
                    [VectConst],[RelaVari],[RelaConst],[FuncConst],[FormVari])
   freeVars :: a -> ([CatObjVar],[ElemVari],[VectVari],[RelaVari])

instance Typed CatObjVar where
   isWellFormed = \ _ -> True
   syntMat = syntMatUsedInCatObjVar
   freeVars = \x -> ([x],[],[],[])
instance Typed CatObject where
   isWellFormed = catObjIsWellFormed
   syntMat = syntMatUsedInCatObj
   freeVars = freeVarInCatObj
instance Typed ElemConst where
   dom = domEC
   isWellFormed = elemConstIsWellFormed
   syntMat = syntMatUsedInElemConst
   freeVars = freeVarInElemCnst
instance Typed ElemVari  where
   dom = domEV
   isWellFormed = elemVariIsWellFormed
   syntMat = syntMatUsedInElemVari
   freeVars = freeVarInElemVari
instance Typed VectConst where
   dom = domVC
   isWellFormed = vectConstIsWellFormed
   syntMat = syntMatUsedInVectConst
   freeVars = freeVarInVectConst
instance Typed VectVari  where
   dom = domVV
   isWellFormed = vectVariIsWellFormed
   syntMat = syntMatUsedInVectVari
   freeVars = freeVarInVectVari
instance Typed RelaConst where
   dom = domRC
   cod = codRC
   typeOf = typeOfRC
```

```
   isWellFormed = relaConstIsWellFormed
   syntMat = syntMatUsedInRelaConst
   freeVars = freeVarInRelaConst
instance Typed FuncConst where
   dom = domFC
   cod = codFC
   isWellFormed = funcConstIsWellFormed
   syntMat = syntMatUsedInFuncConst
   freeVars = freeVarInFuncConst
instance Typed RelaVari  where
   dom = domRV
   cod = codRV
   typeOf = typeOfRV
   isWellFormed = relaVariIsWellFormed
   syntMat = syntMatUsedInRelaVari
   freeVars = freeVarInRelaVari
instance Typed ElemTerm  where
   dom = domET
   isWellFormed = elemTermIsWellFormed
   syntMat = syntMatUsedInElemTerm
   freeVars = freeVarInElemTerm
instance Typed VectTerm  where
   dom = domVT
   isWellFormed = vectTermIsWellFormed
   syntMat = syntMatUsedInVectTerm
   freeVars = freeVarInVectTerm
instance Typed RelaTerm  where
   dom = domRT
   cod = codRT
   typeOf = typeOfRT
   isWellFormed = relaTermIsWellFormed
   syntMat = syntMatUsedInRelaTerm
   freeVars = freeVarInRelaTerm
instance Typed ElemFct   where
   dom = domFE
   isWellFormed = elemFCTIsWellFormed
   syntMat = syntMatUsedInElemFct
   --freeVars = freeVarInElemFCT
instance Typed VectFct   where
   dom = domFV
   typeOf = typeOfFV
   isWellFormed = vectFCTIsWellFormed
   syntMat = syntMatUsedInVectFct
   freeVars = freeVarInVectFct
instance Typed RelaFct   where
   dom = domFR
   cod = codFR
   isWellFormed = relaFCTIsWellFormed
   syntMat = syntMatUsedInRelaFct
   freeVars = freeVarInRelaFct
```

```
instance Typed ElemSET   where
   dom = domES
   isWellFormed = elemSetIsWellFormed
   syntMat = syntMatUsedInElemSet
   freeVars = freeVarInElemSet
instance Typed VectSET   where
   dom = domVS
   isWellFormed = vectSetIsWellFormed
   syntMat = syntMatUsedInVectSet
   freeVars = freeVarInVectSet
instance Typed RelaSET   where
   dom = domRS
   cod = codRS
   typeOf = typeOfRS
   isWellFormed = relaSetIsWellFormed
   syntMat = syntMatUsedInRelaSet
   freeVars = freeVarInRelaSet
instance Typed ElemForm  where
   dom = domEF
   isWellFormed = elemFormIsWellFormed
   syntMat = syntMatUsedInElemForm
   freeVars = freeVarInElemForm
instance Typed VectForm  where
   dom = domVF
   isWellFormed = vectFormIsWellFormed
   syntMat = syntMatUsedInVectForm
   freeVars = freeVarInVectForm
instance Typed RelaForm  where
   dom = domRF
   cod = codRF
   typeOf = typeOfRF
   isWellFormed = relaFormIsWellFormed
   syntMat = syntMatUsedInRelaForm
   freeVars = freeVarInRelaForm
instance Typed FormVari  where
   isWellFormed = formVariIsWellFormed
   syntMat = syntMatUsedInFormVari
   freeVars = freeVarInFormVari
instance Typed Formula   where
   isWellFormed = formulaIsWellFormed
   syntMat = syntMatUsedInFormula
   freeVars = freeVarInFormula
```

For relations it is often more comfortable to denote their arity directly, giving domain and codomain as a pair.

```
typeOfRC :: RelaConst -> (CatObject,CatObject)
typeOfRC (Rela _ o o') = (o,o')

typeOfRV :: RelaVari -> (CatObject,CatObject)
```

```
typeOfRV rv =
   case rv of
      VarR          _   o o' -> (o,o')
      IndexedVarR _ _ o o' -> (o,o')

typeOfRT :: RelaTerm -> (CatObject,CatObject)
typeOfRT rt =
   case rt of
      RC rc -> typeOfRC rc
      RV rv -> typeOfRV  rv
      rt1 :***: rt2  -> (domRT rt1, codRT rt2)
      rt1 :|||: _    -> typeOfRT rt1
      rt1 :&&&: _    -> typeOfRT rt1
      NegaR      rt1 -> typeOfRT rt1
      Ident      d   -> (d,d)
      NullR      d c -> (d,c)
      UnivR      d c -> (d,c)
      Convs      rt1 -> (b,a)    where (a,b) = typeOfRT rt1
      vt1 :||--: vt2 -> (domVT vt1, domVT vt2)
      SupRela    rs  -> typeOfRS rs
      InfRela    rs  -> typeOfRS rs
      (:*:)    _ _   -> typeOfRT $ expandDefinedRelaTerm rt
      (:\/:)   _ _   -> typeOfRT $ expandDefinedRelaTerm rt
      Pi       o o'  -> (DirPro o o',o)
      Rho      o o'  -> (DirPro o o',o')
      Iota     o o'  -> (o ,DirSum o o')
      Kappa    o o'  -> (o',DirSum o o')
      CASE rt1 rt2   -> (DirSum (domRT rt1) (domRT rt2), codRT rt1)
      Project  rt2   -> (domRT rt2,QuotMod rt2)
      Epsi     o     -> (o,DirPow o)
      PointDiag  et  -> (a,a)    where a = domET et
      SyQ rt1 rt2    -> typeOfRT $ expandDefinedRelaTerm rt
      --Wait      rt1  -> (Strict (domPT rt1),Strict (codPT rt1))
      Belly pt -> (domRT $ PartOrd $ domPT pt,codRT $ PartOrd $ codPT pt)
      InjTerm    vt    -> (InjFrom vt, domVT vt)
      ProdVectToRela _ -> typeOfRT $ expandDefinedRelaTerm rt
      PartOrd _        -> (Lifted $ domRT rt,Lifted $ codRT rt)
      RFctAppl rf rt -> snd $ typeOfFR rf

typeOfPT :: PartTerm -> (ParObject,ParObject)
typeOfPT pt =
   case pt of
      Lift   rt1      -> (ParObj (domRT rt1),ParObj (codRT rt1))
      Fetus po1 _ po2 -> (po1,po2)
      pt1 :*****: pt2 -> (domPT pt1, codPT pt2)
      pt1 :|||||: _   -> typeOfPT pt1
      pt1 :&&&&&: _   -> typeOfPT pt1
      NegaPart pt1    -> typeOfPT pt1
      IdentP po       -> (po,po)
      NullP  d c      -> (d,c)
```

```
      UnivP  d c      -> (d,c)
      TranspP pt1     -> (codPT pt1,domPT pt1)
      PPi    o o'     -> (ParPro o o',o)
      PRho   o o'     -> (ParPro o o',o')
      pt1 :#: pt2     -> (ParPro (domPT pt1) (domPT pt2),
                          ParPro (codPT pt1) (codPT pt2))
      pt1 :\\//: pt2  -> (domPT pt1,
                          ParPro (codPT pt1) (codPT pt2))
      PIota  o o'     -> (o ,ParPro o o')
      PKappa o o'     -> (o',ParPro o o')
      PEpsi  o        -> (o ,ParPow o)
typeOfPF :: PartForm -> (ParObject,ParObject)
typeOfPF rf =
    case rf of
      rt1 :<====: rt2   -> typeOfPT rt1
      rt1 :>====: rt2   -> typeOfPT rt1
      rt1 :<=/==: rt2   -> typeOfPT rt1
      rt1 :>=/==: rt2   -> typeOfPT rt1
      StrictPartContained pt1 pt2 -> typeOfPT pt1


typeOfRS rs =
   case rs of
      VarRS _ o o' -> (o,o')
      RS rv  _     -> (domRV rv,codRV rv)
      RT f   _     -> snd $ typeOfFR f
      RX rts o o'  -> (o,o')

typeOfRF :: RelaForm -> (CatObject,CatObject)
typeOfRF rf =
    case rf of
      rt1 :<==: _ -> typeOfRT rt1
      rt1 :>==: _ -> typeOfRT rt1
      rt1 :===: _ -> typeOfRT rt1
      rt1 :<=/: _ -> typeOfRT rt1
      rt1 :>=/: _ -> typeOfRT rt1
      rt1 :=/=: _ -> typeOfRT rt1
      RelaInSet rt _ -> typeOfRT rt
      REE rt _ _     -> typeOfRT rt
      QuantRelaForm _ rv _ -> typeOfRV rv

typeOfFV :: VectFct -> (CatObject,CatObject)
typeOfFV (VFCT vv vt) = (domVV vv,domVT vt)

typeOfFR :: RelaFct -> ((CatObject,CatObject),(CatObject,CatObject))
typeOfFR (RFCT evr rt) =
   case evr of
      EVar ea -> ((domEV ea,domEV ea),typeOfRT rt)
      VVar va -> ((domVV va,domVV va),typeOfRT rt)
      RVar ra -> (typeOfRV ra,typeOfRT rt)
```

### 2.4.2 Inductive Well-Formedness Definitions

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

```
catObjIsWellFormed :: CatObject -> Bool
catObjIsWellFormed co =
   case co of
      OC (CstO _) -> True
      OV (VarO _) -> True
      OV (IndexedVarO _ _) -> True
      --Strict po   -> parObjIsWellFormed po
      DirPro o o' -> catObjIsWellFormed o && catObjIsWellFormed o'
      DirSum o o' -> catObjIsWellFormed o && catObjIsWellFormed o'
      DirPow o    -> catObjIsWellFormed o
      UnitOb      -> True
      InjFrom  vt -> vectTermIsWellFormed vt
      QuotMod  rt -> relaTermIsWellFormed rt
      Strict   po -> parObjIsWellFormed   po

parObjIsWellFormed :: ParObject -> Bool
parObjIsWellFormed po =
   case po of
      ParObj co   -> catObjIsWellFormed co
      ParPro o o' -> parObjIsWellFormed o && parObjIsWellFormed o'
      ParSum o o' -> parObjIsWellFormed o && parObjIsWellFormed o'
      ParPow o    -> parObjIsWellFormed o


elemConstIsWellFormed :: ElemConst -> Bool
elemConstIsWellFormed ec =
    case ec of
       Elem                _ o -> catObjIsWellFormed o
       GenericElemNotation o _ -> catObjIsWellFormed o
elemVariIsWellFormed :: ElemVari -> Bool
elemVariIsWellFormed et =
    case et of
       VarE          _ o -> catObjIsWellFormed o
       IndexedVarE _ _ o -> catObjIsWellFormed o

elemTermIsWellFormed :: ElemTerm -> Bool
elemTermIsWellFormed et =
   case et of
      EC ec -> elemConstIsWellFormed ec
      EV ev -> elemVariIsWellFormed  ev
      Pair et1 et2 -> elemTermIsWellFormed et1  &&
                      elemTermIsWellFormed et2
      Inj1 et  o   -> elemTermIsWellFormed et   &&
                      catObjIsWellFormed   o
      Inj2 o   et  -> catObjIsWellFormed   o    &&
                      elemTermIsWellFormed et
      ThatV vt -> vectTermIsWellFormed vt
```

```
      SomeV vt -> vectTermIsWellFormed vt
      ThatR rt -> relaTermIsWellFormed rt
      SomeR rt -> relaTermIsWellFormed rt
      FuncAppl ef et -> elemTermIsWellFormed  et &&
                        funcConstIsWellFormed ef &&
                           (domFC ef == domET et)
      VectToElem _ -> elemTermIsWellFormed $ expandDefinedElemTerm et

elemFCTIsWellFormed :: ElemFct -> Bool
elemFCTIsWellFormed (EFCT ev et) = elemVariIsWellFormed ev &&
                                   elemTermIsWellFormed et

vectConstIsWellFormed :: VectConst -> Bool
vectConstIsWellFormed vc =
    case vc of
       Vect _  o -> catObjIsWellFormed o

vectVariIsWellFormed :: VectVari -> Bool
vectVariIsWellFormed vv =
   case vv of
      VarV          _ o -> catObjIsWellFormed o
      IndexedVarV _ _ o -> catObjIsWellFormed o

vectTermIsWellFormed :: VectTerm -> Bool
vectTermIsWellFormed vt =
   case vt of
      VC vc          -> vectConstIsWellFormed vc
      VV vv          -> vectVariIsWellFormed  vv
      rt  :****: vt2 -> relaTermIsWellFormed rt  &&
                        vectTermIsWellFormed vt2 &&
                           (codRT rt  == domVT vt2)
      vt1 :||||: vt2 -> vectTermIsWellFormed vt1 &&
                        vectTermIsWellFormed vt2 &&
                           (domVT vt1 == domVT vt2)
      vt1 :&&&&: vt2 -> vectTermIsWellFormed vt1 &&
                        vectTermIsWellFormed vt2 &&
                           (domVT vt1 == domVT vt2)
      Syq rt vt2     -> relaTermIsWellFormed rt  &&
                        vectTermIsWellFormed vt2 &&
                           (domRT rt  == domVT vt2)
      NegaV      vt1 -> vectTermIsWellFormed vt1
      NullV      o   -> catObjIsWellFormed o
      UnivV      o   -> catObjIsWellFormed o
      SupVect    vs  -> vectSetIsWellFormed  vs
      InfVect    vs  -> vectSetIsWellFormed  vs
      PointVect  et  -> elemTermIsWellFormed et
      RelaToVect  _  -> vectTermIsWellFormed $ expandDefinedVectTerm vt
      VFctAppl vf vt2 -> let (a,_) = typeOfFV vf
                             c     = domVT vt2
                         in  vectFCTIsWellFormed vf &&
```

```
                              vectTermIsWellFormed vt2 && (c == a)
      PowElemToVect et -> elemTermIsWellFormed et

vectFCTIsWellFormed :: VectFct -> Bool
vectFCTIsWellFormed (VFCT vv vt) = vectVariIsWellFormed vv &&
                                   vectTermIsWellFormed vt


relaConstIsWellFormed :: RelaConst -> Bool
relaConstIsWellFormed rc =
    case rc of
       Rela _ o o' ->  catObjIsWellFormed o && catObjIsWellFormed o'

funcConstIsWellFormed :: FuncConst -> Bool
funcConstIsWellFormed fc =
    case fc of
       Func _ o o' ->  catObjIsWellFormed o && catObjIsWellFormed o'

relaVariIsWellFormed :: RelaVari -> Bool
relaVariIsWellFormed rt =
   case rt of
      VarR        _   o o' -> catObjIsWellFormed o && catObjIsWellFormed o'
      IndexedVarR _ _ o o' -> catObjIsWellFormed o && catObjIsWellFormed o'

relaTermIsWellFormed :: RelaTerm -> Bool
relaTermIsWellFormed rt =
   case rt of
      RC rc            -> relaConstIsWellFormed rc
      RV rv            -> relaVariIsWellFormed rv
      rt1 :***: rt2    -> relaTermIsWellFormed rt1 &&
                          relaTermIsWellFormed rt2 && (codRT rt1 == domRT rt2)
      rt1 :|||: rt2    -> relaTermIsWellFormed rt1 &&
                          relaTermIsWellFormed rt2 && (typeOfRT rt1 == typeOfRT rt2)
      rt1 :&&&: rt2    -> relaTermIsWellFormed rt1 &&
                          relaTermIsWellFormed rt2 && (typeOfRT rt1 == typeOfRT rt2)
      NegaR     rt1    -> relaTermIsWellFormed rt1
      Ident     d      -> catObjIsWellFormed d
      NullR     d c    -> catObjIsWellFormed d && catObjIsWellFormed c
      UnivR     d c    -> catObjIsWellFormed d && catObjIsWellFormed c
      Convs     rt1    -> relaTermIsWellFormed rt1
      vt1 :||--: vt2   -> vectTermIsWellFormed vt1 && vectTermIsWellFormed vt2
      SupRela   rs     -> relaSetIsWellFormed  rs
      InfRela   rs     -> relaSetIsWellFormed  rs
      (:*:)     _ _    -> relaTermIsWellFormed $ expandDefinedRelaTerm rt
      (:\/:)    _ _    -> relaTermIsWellFormed $ expandDefinedRelaTerm rt
      Pi        o o'   -> catObjIsWellFormed o && catObjIsWellFormed o'
      Rho       o o'   -> catObjIsWellFormed o && catObjIsWellFormed o'
      Iota      o o'   -> catObjIsWellFormed o && catObjIsWellFormed o'
      Kappa     o o'   -> catObjIsWellFormed o && catObjIsWellFormed o'
      CASE rt1 rt2     -> relaTermIsWellFormed rt1 &&
```

```
                        relaTermIsWellFormed rt2 && (codRT rt1 == codRT rt2)
      Project    rt1   -> relaTermIsWellFormed rt1
      Epsi     o       -> catObjIsWellFormed o
      PointDiag et     -> elemTermIsWellFormed et
      SyQ      _ _     -> relaTermIsWellFormed $ expandDefinedRelaTerm rt
      --Wait      rt1    -> partTermIsWellFormed rt1
      Belly pt         -> partTermIsWellFormed pt
      InjTerm   vt     -> vectTermIsWellFormed vt
      ProdVectToRela _ -> relaTermIsWellFormed $ expandDefinedRelaTerm rt
      PartOrd po       -> parObjIsWellFormed po
      RFctAppl  rf rt2 -> let ((a,b),_) = typeOfFR rf
                          in  relaFCTIsWellFormed rf && evrTermIsWellFormed rt2
evrTermIsWellFormed :: ArgEVR -> Bool
evrTermIsWellFormed evr =
   case evr of
      ArgE ae -> elemTermIsWellFormed ae
      ArgV av -> vectTermIsWellFormed av
      ArgR ar -> relaTermIsWellFormed ar


partTermIsWellFormed :: PartTerm -> Bool
partTermIsWellFormed pt =
   case pt of
      Lift   rt1      -> relaTermIsWellFormed rt1
      Fetus po1 rt1 po2 -> parObjIsWellFormed po1 && parObjIsWellFormed po2 &&
                           relaTermIsWellFormed rt1 &&
                          (domRT $ PartOrd po1, domRT $ PartOrd po2) == typeOfRT rt1
      pt1 :*****: pt2 -> partTermIsWellFormed pt1 && partTermIsWellFormed pt2 &&
                              (codPT pt1 == (domPT pt2))
      pt1 :|||||: pt2 -> partTermIsWellFormed pt1 && partTermIsWellFormed pt2 &&
                              (typeOfPT pt1 == typeOfPT pt2)
      pt1 :&&&&&: pt2 -> partTermIsWellFormed pt1 && partTermIsWellFormed pt2 &&
                              (typeOfPT pt1 == typeOfPT pt2)
      NegaPart pt1    -> partTermIsWellFormed pt1
      IdentP  po      -> parObjIsWellFormed po
      NullP   d c     -> parObjIsWellFormed d && parObjIsWellFormed c
      UnivP   d c     -> parObjIsWellFormed d && parObjIsWellFormed c
      TranspP pt1     -> partTermIsWellFormed pt1
      PPi     o o'    -> parObjIsWellFormed o && parObjIsWellFormed o'
      PRho    o o'    -> parObjIsWellFormed o && parObjIsWellFormed o'
      pt1 :#: pt2     -> partTermIsWellFormed pt1 && partTermIsWellFormed pt2
      pt1 :\\//: pt2  -> partTermIsWellFormed pt1 && partTermIsWellFormed pt2 &&
                              (domPT pt1 == (domPT pt2))
      PIota   o o'    -> parObjIsWellFormed o && parObjIsWellFormed o'
      PKappa  o o'    -> parObjIsWellFormed o && parObjIsWellFormed o'
      PEpsi   o       -> parObjIsWellFormed o

relaFCTIsWellFormed :: RelaFct -> Bool
relaFCTIsWellFormed (RFCT evr rt) =
   case evr of
```

```
      EVar ea -> elemVariIsWellFormed ea && relaTermIsWellFormed rt
      VVar va -> vectVariIsWellFormed va && relaTermIsWellFormed rt
      RVar ra -> relaVariIsWellFormed ra && relaTermIsWellFormed rt


elemSetIsWellFormed es =
   case es of
      VarES _ o -> catObjIsWellFormed o
      ES ev  fs -> and (map formulaIsWellFormed  fs ) && elemVariIsWellFormed ev
      ET     o  -> catObjIsWellFormed o
      EX ets o  -> and (map elemTermIsWellFormed ets) &&
                   and (map (\x -> domET x == o) ets) && catObjIsWellFormed o


vectSetIsWellFormed vs =
   case vs of
      VarVS _ o -> catObjIsWellFormed o
      VS vv  fs -> and (map formulaIsWellFormed  fs ) && vectVariIsWellFormed vv
      VX vts o  -> and (map vectTermIsWellFormed vts) &&
                   and (map (\x -> domVT x == o) vts) && catObjIsWellFormed o


relaSetIsWellFormed rs =
   case rs of
      VarRS _ o o'  -> catObjIsWellFormed o && (catObjIsWellFormed o')
      RS rv  fs     -> and (map formulaIsWellFormed  fs)  && relaVariIsWellFormed rv
      RT f   (ET o) -> relaFCTIsWellFormed f && catObjIsWellFormed o &&
                       ((fst $ fst $ typeOfFR f) == o)
      RX rts o o'   -> and (map relaTermIsWellFormed rts) &&
                       and (map (\x -> typeOfRT x == (o, o')) rts) &&
                       catObjIsWellFormed o && catObjIsWellFormed o'


elemFormIsWellFormed :: ElemForm -> Bool
elemFormIsWellFormed f =
   case f of
     Equation et1 et2    -> elemTermIsWellFormed et1 &&
                            elemTermIsWellFormed et2 && (domET et1 == domET et2)
     NegaEqua et1 et2    -> elemTermIsWellFormed et1 &&
                            elemTermIsWellFormed et2 && (domET et1 == domET et2)
     QuantElemForm _ ev f' -> formulaeAreWellFormed f' && elemVariIsWellFormed ev


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

```
                        (domVT vt1 == domVT vt2)
     vt1 :<=/=: vt2  -> vectTermIsWellFormed vt1 && vectTermIsWellFormed vt2 &&
                        (domVT vt1 == domVT vt2)
     vt1 :>=/=: vt2  -> vectTermIsWellFormed vt1 && vectTermIsWellFormed vt2 &&
                        (domVT vt1 == domVT vt2)
     vt1 :==/=: vt2  -> vectTermIsWellFormed vt1 && vectTermIsWellFormed vt2 &&
                        (domVT vt1 == domVT vt2)
     VE  vt et       -> vectTermIsWellFormed vt  && elemTermIsWellFormed et &&
                        (domET et == domVT vt)
     VectInSet vt vs -> vectTermIsWellFormed vt  && vectSetIsWellFormed  vs  &&
                        (domVT vt == domVS vs)
     QuantVectForm _ vv fs -> formulaeAreWellFormed fs && vectVariIsWellFormed vv


relaFormIsWellFormed :: RelaForm -> Bool
relaFormIsWellFormed rf =
    case rf of
     rt1 :<==: rt2   -> relaTermIsWellFormed rt1 && relaTermIsWellFormed rt2 &&
                        (typeOfRT rt1 == typeOfRT rt2)
     rt1 :>==: rt2   -> relaTermIsWellFormed rt1 && relaTermIsWellFormed rt2 &&
                        (typeOfRT rt1 == typeOfRT rt2)
     rt1 :===: rt2   -> relaTermIsWellFormed rt1 && relaTermIsWellFormed rt2 &&
                        (typeOfRT rt1 == typeOfRT rt2)
     rt1 :<=/: rt2   -> relaTermIsWellFormed rt1 && relaTermIsWellFormed rt2 &&
                        (typeOfRT rt1 == typeOfRT rt2)
     rt1 :>=/: rt2   -> relaTermIsWellFormed rt1 && relaTermIsWellFormed rt2 &&
                        (typeOfRT rt1 == typeOfRT rt2)
     rt1 :=/=: rt2   -> relaTermIsWellFormed rt1 && relaTermIsWellFormed rt2 &&
                        (typeOfRT rt1 == typeOfRT rt2)
     REE rt et1 et2  -> let (o1,o2) = typeOfRT rt
                        in  relaTermIsWellFormed rt  &&
                            elemTermIsWellFormed et1 && (domET et1 == o1) &&
                            elemTermIsWellFormed et2 && (domET et2 == o2)
     RelaInSet rt rs -> relaTermIsWellFormed rt && relaSetIsWellFormed rs &&
                        (typeOfRT rt == typeOfRS rs)
     QuantRelaForm _ rv fs -> formulaeAreWellFormed fs && relaVariIsWellFormed rv
relaFormsAreWellFormed :: [RelaForm] -> Bool
relaFormsAreWellFormed rfs = and (map relaFormIsWellFormed rfs)

partFormIsWellFormed :: PartForm -> Bool
partFormIsWellFormed rf =
    case rf of
      rt1 :<====: rt2   -> partTermIsWellFormed rt1 && partTermIsWellFormed rt2 &&
                           (typeOfPT rt1 == typeOfPT rt2)
      rt1 :>====: rt2   -> partTermIsWellFormed rt1 && partTermIsWellFormed rt2 &&
                           (typeOfPT rt1 == typeOfPT rt2)
      rt1 :<=/==: rt2   -> partTermIsWellFormed rt1 && partTermIsWellFormed rt2 &&
                           (typeOfPT rt1 == typeOfPT rt2)
      rt1 :>=/==: rt2   -> partTermIsWellFormed rt1 && partTermIsWellFormed rt2 &&
                           (typeOfPT rt1 == typeOfPT rt2)
```

```
      StrictPartContained pt1 pt2 ->
                          partTermIsWellFormed pt1 && partTermIsWellFormed pt2 &&
                          (typeOfPT pt1 == typeOfPT pt2)


formVariIsWellFormed :: FormVari -> Bool
formVariIsWellFormed fv =
   case fv of
      VarF          _ -> True
      IndexedVarF _ _ -> True

formulaIsWellFormed :: Formula -> Bool
formulaIsWellFormed f =
   case f of
      EF ef          -> elemFormIsWellFormed ef
      VF vf          -> vectFormIsWellFormed vf
      RF rf          -> relaFormIsWellFormed rf
      --PF pf          -> partFormIsWellFormed pf
      Verum          -> True
      Falsum         -> True
      Negated  f1    -> formulaIsWellFormed  f1
      Implies  f1 f2 -> formulaIsWellFormed  f1 && formulaIsWellFormed f2
      SemEqu   f1 f2 -> formulaIsWellFormed  f1 && formulaIsWellFormed f2
      Disjunct f1 f2 -> formulaIsWellFormed  f1 && formulaIsWellFormed f2
      Conjunct f1 f2 -> formulaIsWellFormed  f1 && formulaIsWellFormed f2

formulaeAreWellFormed :: [Formula] -> Bool
formulaeAreWellFormed fs = and (map formulaIsWellFormed fs)
```

### 2.4.3 Inductive Collection of Syntactical Material

The syntactical material may be collected accumulating the category object variables, category object constants, element variables, element constants, vector variables, vector constants, relation variables, relation constants, function variables, and function constants, in this sequence. A basic function allows to add syntactical material collected in different constituents of a term.

```
nP u v = nub (u ++ v)
joinSyntMat (a,b,c,d,e,f,g,h,i,j) (u,v,w,x,q,r,s,t,t',u') = (nP a u, nP b v,
            nP c w, nP d x, nP e q, nP f r, nP g s, nP h t, nP i t', nP j u')
joinSyntMatSet s = foldl joinSyntMat ([],[],[],[],[],[],[],[],[],[]) s

showSyntMat (a,b,c,d,e,f,g,h,i,j) =
   "ObjVari  " ++ show a ++ "\n" ++
   "ObjConst " ++ show b ++ "\n" ++
   "ElemVari " ++ show c ++ "\n" ++
   "ElemConst" ++ show d ++ "\n" ++
   "VectVari " ++ show e ++ "\n" ++
   "VectConst" ++ show f ++ "\n" ++
   "RelaVari " ++ show g ++ "\n" ++
```

```
   "RelaConst" ++ show h ++ "\n" ++
   "FuncConst" ++ show h ++ "\n" ++
   "FormVari"  ++ show i ++ "\n"


syntMatUsedInCatObjVar :: CatObjVar ->
    ([CatObjVar],[CatObjCst],[ElemVari],[ElemConst],
     [VectVari ],[VectConst],[RelaVari],[RelaConst],[FuncConst],[FormVari])
syntMatUsedInCatObjVar cov = ([cov],[],[],[],[],[],[],[],[],[])

syntMatUsedInCatObj :: CatObject ->
    ([CatObjVar],[CatObjCst],[ElemVari],[ElemConst],
     [VectVari ],[VectConst],[RelaVari],[RelaConst],[FuncConst],[FormVari])
syntMatUsedInCatObj co =
   case co of
      OC oc -> ([],[oc],[],[],[],[],[],[],[],[])
      OV ov -> syntMatUsedInCatObjVar ov
      --Strict po   -> syntMatUsedInParObj po
      DirPro o o' -> joinSyntMat (syntMatUsedInCatObj o)
                                 (syntMatUsedInCatObj o')
      DirSum o o' -> joinSyntMat (syntMatUsedInCatObj o)
                                 (syntMatUsedInCatObj o')
      DirPow o    -> syntMatUsedInCatObj o
      UnitOb      -> ([],[],[],[],[],[],[],[],[],[])
      InjFrom vt  -> syntMatUsedInVectTerm vt
      QuotMod rt  -> syntMatUsedInRelaTerm rt
      Strict  po  -> syntMatUsedInParObj   po

syntMatUsedInCatObjs cos = joinSyntMatSet (map syntMatUsedInCatObj cos)

syntMatUsedInParObj po =
   case po of
      ParObj co   -> syntMatUsedInCatObj co
      ParPro o o' -> joinSyntMat (syntMatUsedInParObj o)
                                 (syntMatUsedInParObj o')
      ParSum o o' -> joinSyntMat (syntMatUsedInParObj o)
                                 (syntMatUsedInParObj o')
      ParPow o    -> syntMatUsedInParObj o



syntMatUsedInElemConst ec@(Elem _ o) =
   joinSyntMat (syntMatUsedInCatObj o) ([],[],[],[ec],[],[],[],[],[],[])


syntMatUsedInElemVari ev =
   case ev of
      VarE          _   o -> joinSyntMat (syntMatUsedInCatObj o)
                                         ([],[],[ev],[],[],[],[],[],[],[])
      IndexedVarE _ _ o -> joinSyntMat (syntMatUsedInCatObj o)
                                         ([],[],[ev],[],[],[],[],[],[],[])
```

```
syntMatUsedInElemVaris evs =  joinSyntMatSet (map syntMatUsedInElemVari evs)

syntMatUsedInElemTerm et =
   case et of
      EC    ec -> syntMatUsedInElemConst ec
      EV    ev -> syntMatUsedInElemVari  ev
      Pair et1 et2 -> joinSyntMat (syntMatUsedInElemTerm et1)
                                  (syntMatUsedInElemTerm et2)
      Inj1 et o    -> joinSyntMat (syntMatUsedInCatObj   o)
                                  (syntMatUsedInElemTerm et)
      Inj2 o  et   -> joinSyntMat (syntMatUsedInCatObj   o)
                                  (syntMatUsedInElemTerm et)
      ThatV vt -> syntMatUsedInVectTerm  vt
      SomeV vt -> syntMatUsedInVectTerm  vt
      ThatR rt -> syntMatUsedInRelaTerm  rt
      SomeR rt -> syntMatUsedInRelaTerm  rt
      FuncAppl fc et -> joinSyntMat ([],[],[],[],[],[],[],[],[fc],[])
                                    (syntMatUsedInElemTerm et)
      VectToElem  vt -> syntMatUsedInElemTerm $ expandDefinedElemTerm et
syntMatUsedInElemTerms ets = joinSyntMatSet (map syntMatUsedInElemTerm ets)


syntMatUsedInVectConst vc@(Vect _ o) =
   joinSyntMat (syntMatUsedInCatObj o) ([],[],[],[],[],[vc],[],[],[],[])

syntMatUsedInVectVari vv =
   case vv of
      VarV          _ o -> joinSyntMat (syntMatUsedInCatObj o)
                                       ([],[],[],[],[vv],[],[],[],[],[])
      IndexedVarV _ _ o -> joinSyntMat (syntMatUsedInCatObj o)
                                       ([],[],[],[],[vv],[],[],[],[],[])

syntMatUsedInVectTerm vt =
   case vt of
      VC vc           -> syntMatUsedInVectConst vc
      VV vv           -> syntMatUsedInVectVari  vv
      rt1 :****: vt2  -> joinSyntMat (syntMatUsedInRelaTerm rt1)
                                     (syntMatUsedInVectTerm vt2)
      vt1 :||||: vt2  -> joinSyntMat (syntMatUsedInVectTerm vt1)
                                     (syntMatUsedInVectTerm vt2)
      vt1 :&&&&: vt2  -> joinSyntMat (syntMatUsedInVectTerm vt1)
                                     (syntMatUsedInVectTerm vt2)
      Syq rt1 vt2     -> joinSyntMat (syntMatUsedInRelaTerm rt1)
                                     (syntMatUsedInVectTerm vt2)
      NegaV      vt1  -> syntMatUsedInVectTerm vt1
      NullV      o    -> syntMatUsedInCatObj   o
      UnivV      o    -> syntMatUsedInCatObj   o
      SupVect    vs   -> syntMatUsedInVectSet  vs
      InfVect    vs   -> syntMatUsedInVectSet  vs
      RelaToVect _    -> syntMatUsedInVectTerm $ expandDefinedVectTerm vt
```

```
      PointVect  et   -> syntMatUsedInElemTerm et
      VFctAppl vf vt2 -> joinSyntMat (syntMatUsedInVectFct  vf)
                                     (syntMatUsedInVectTerm vt2)
      PowElemToVect et -> syntMatUsedInElemTerm et


syntMatUsedInVectTerms vts = joinSyntMatSet (map syntMatUsedInVectTerm vts)

syntMatUsedInRelaConst rc@(Rela _ o o') =
   joinSyntMatSet [syntMatUsedInCatObj o,
                   syntMatUsedInCatObj o', ([],[],[],[],[],[],[],[rc],[],[])]

syntMatUsedInFuncConst fc@(Func _ o o') =
   joinSyntMatSet [syntMatUsedInCatObj o,
                   syntMatUsedInCatObj o', ([],[],[],[],[],[],[],[],[fc],[])]

syntMatUsedInRelaVari rv =
   case rv of
      VarR          _ o o' -> joinSyntMatSet [syntMatUsedInCatObj o,
                                              syntMatUsedInCatObj o',
                                              ([],[],[],[],[],[],[rv],[],[],[])]
      IndexedVarR _ _ o o' -> joinSyntMatSet [syntMatUsedInCatObj o,
                                              syntMatUsedInCatObj o',
                                              ([],[],[],[],[],[],[rv],[],[],[])]


syntMatUsedInRelaTerm rt =
   case rt of
      RC rc          -> syntMatUsedInRelaConst rc
      RV rv          -> syntMatUsedInRelaVari  rv
      rt1 :***:  rt2 -> joinSyntMat (syntMatUsedInRelaTerm rt1)
                                    (syntMatUsedInRelaTerm rt2)
      rt1 :|||:  rt2 -> joinSyntMat (syntMatUsedInRelaTerm rt1)
                                    (syntMatUsedInRelaTerm rt2)
      rt1 :&&&:  rt2 -> joinSyntMat (syntMatUsedInRelaTerm rt1)
                                    (syntMatUsedInRelaTerm rt2)
      NegaR      rt1 -> syntMatUsedInRelaTerm rt1
      Ident      o   -> syntMatUsedInCatObj o
      NullR      d c -> joinSyntMat (syntMatUsedInCatObj c)
                                    (syntMatUsedInCatObj d)
      UnivR      d c -> joinSyntMat (syntMatUsedInCatObj c)
                                    (syntMatUsedInCatObj d)
      Convs      rt1 -> syntMatUsedInRelaTerm rt1
      vt1 :||--: vt2 -> joinSyntMat (syntMatUsedInVectTerm vt1)
                                    (syntMatUsedInVectTerm vt2)
      SupRela    rs  -> syntMatUsedInRelaSet  rs
      InfRela    rs  -> syntMatUsedInRelaSet  rs
      (:*:)    _ _   -> syntMatUsedInRelaTerm $ expandDefinedRelaTerm rt
      (:\/:)   _ _   -> syntMatUsedInRelaTerm $ expandDefinedRelaTerm rt
      Pi       o o'  -> joinSyntMat (syntMatUsedInCatObj o)
                                    (syntMatUsedInCatObj o')
```

```
      Rho      o o'  -> joinSyntMat (syntMatUsedInCatObj o)
                                    (syntMatUsedInCatObj o')
      Iota     o o'  -> joinSyntMat (syntMatUsedInCatObj o)
                                    (syntMatUsedInCatObj o')
      Kappa    o o'  -> joinSyntMat (syntMatUsedInCatObj o)
                                    (syntMatUsedInCatObj o')
      CASE rt1 rt2   -> joinSyntMat (syntMatUsedInRelaTerm rt1)
                                    (syntMatUsedInRelaTerm rt2)
      Project  rt1   -> syntMatUsedInRelaTerm rt1
      Epsi     o     -> syntMatUsedInCatObj o
      PointDiag  et  -> syntMatUsedInElemTerm et
      SyQ      _ _   -> syntMatUsedInRelaTerm $ expandDefinedRelaTerm rt
      --Wait       rt1 -> syntMatUsedInPartTerm rt1
      Belly pt         -> syntMatUsedInPartTerm pt
      InjTerm    vt    -> syntMatUsedInVectTerm vt
      ProdVectToRela _ -> syntMatUsedInRelaTerm $ expandDefinedRelaTerm rt
      PartOrd po       -> syntMatUsedInParObj po
      RFctAppl rf rt2  -> joinSyntMat (syntMatUsedInRelaFct rf)
                                      (syntMatUsedInEVRTerm rt2)

syntMatUsedInEVRTerm evr =
   case evr of
      ArgE ae -> syntMatUsedInElemTerm ae
      ArgV av -> syntMatUsedInVectTerm av
      ArgR ar -> syntMatUsedInRelaTerm ar

syntMatUsedInPartTerm rt =
   case rt of
      Lift   rt1      -> syntMatUsedInRelaTerm rt1
      Fetus po1 rt1 po2 -> joinSyntMatSet [syntMatUsedInParObj   po1,
                                           syntMatUsedInRelaTerm rt1,
                                           syntMatUsedInParObj   po2]
      pt1 :*****: pt2 -> joinSyntMat (syntMatUsedInPartTerm pt1)
                                     (syntMatUsedInPartTerm pt2)
      pt1 :|||||: pt2 -> joinSyntMat (syntMatUsedInPartTerm pt1)
                                     (syntMatUsedInPartTerm pt2)
      pt1 :&&&&&: pt2 -> joinSyntMat (syntMatUsedInPartTerm pt1)
                                     (syntMatUsedInPartTerm pt2)
      NegaPart pt1    -> syntMatUsedInPartTerm pt1
      IdentP po       -> syntMatUsedInParObj po
      NullP  d c      -> joinSyntMat (syntMatUsedInParObj d)
                                     (syntMatUsedInParObj c)
      UnivP  d c      -> joinSyntMat (syntMatUsedInParObj d)
                                     (syntMatUsedInParObj c)
      TranspP pt1     -> syntMatUsedInPartTerm pt1
      PPi    o o'     -> joinSyntMat (syntMatUsedInParObj o)
                                     (syntMatUsedInParObj o')
      PRho   o o'     -> joinSyntMat (syntMatUsedInParObj o)
                                     (syntMatUsedInParObj o')
      pt1 :#: pt2     -> joinSyntMat (syntMatUsedInPartTerm pt1)
```

```
                                    (syntMatUsedInPartTerm pt2)
      pt1 :\\//: pt2  -> joinSyntMat (syntMatUsedInPartTerm pt1)
                                    (syntMatUsedInPartTerm pt2)
      PIota  o o'     -> joinSyntMat (syntMatUsedInParObj o)
                                    (syntMatUsedInParObj o')
      PKappa o o'     -> joinSyntMat (syntMatUsedInParObj o)
                                    (syntMatUsedInParObj o')
      PEpsi  o        -> syntMatUsedInParObj o


syntMatUsedInElemFct (EFCT ev et) = joinSyntMat (syntMatUsedInElemVari ev)
                                                (syntMatUsedInElemTerm et)
syntMatUsedInVectFct (VFCT vv vt) = joinSyntMat (syntMatUsedInVectVari vv)
                                                (syntMatUsedInVectTerm vt)
syntMatUsedInRelaFct (RFCT evr rt) =
   case evr of
      EVar ea -> joinSyntMat (syntMatUsedInElemVari ea)
                             (syntMatUsedInRelaTerm rt)
      VVar va -> joinSyntMat (syntMatUsedInVectVari va)
                             (syntMatUsedInRelaTerm rt)
      RVar ra -> joinSyntMat (syntMatUsedInRelaVari ra)
                             (syntMatUsedInRelaTerm rt)


syntMatUsedInRelaTerms rts = joinSyntMatSet (map syntMatUsedInRelaTerm rts)

syntMatUsedInElemSet es =
   case es of
      VarES _ o -> syntMatUsedInCatObj o
      ET      o -> syntMatUsedInCatObj o
      ES ev  fs -> joinSyntMat  (syntMatUsedInFormulae fs)
                                (syntMatUsedInElemVari ev)
      EX ets o  -> joinSyntMat (syntMatUsedInElemTerms ets)
                               (syntMatUsedInCatObj o)
syntMatUsedInVectSet vs =
   case vs of
      VarVS _ o -> syntMatUsedInCatObj o
      VS vv fs  -> let (a,b,c,d,e,f,g,h,i,j) = syntMatUsedInFormulae fs
                   in  joinSyntMat (a,b,c,d,e,f,g,h,i,j) (syntMatUsedInVectVari vv)
      VX vts o  -> joinSyntMat (syntMatUsedInVectTerms vts)
                               (syntMatUsedInCatObj o)
syntMatUsedInRelaSet rs =
   case rs of
      VarRS _ o o' -> joinSyntMat (syntMatUsedInCatObj o)
                                  (syntMatUsedInCatObj o')
      RS rv fs     -> joinSyntMat (syntMatUsedInFormulae fs)
                                  (syntMatUsedInRelaVari rv)
      RX rts o o'  -> joinSyntMatSet [syntMatUsedInRelaTerms rts,
                                      syntMatUsedInCatObj o,
                                      syntMatUsedInCatObj o']
```

```
syntMatUsedInFormVari fv = ([],[],[],[],[],[],[],[],[],[fv])

syntMatUsedInElemForm f =
   case f of
     Equation et1 et2         -> joinSyntMat (syntMatUsedInElemTerm et1)
                                             (syntMatUsedInElemTerm et2)
     NegaEqua et1 et2         -> joinSyntMat (syntMatUsedInElemTerm et1)
                                             (syntMatUsedInElemTerm et2)
     QuantElemForm    _ ev fs -> joinSyntMat (syntMatUsedInElemVari ev)
                                             (syntMatUsedInFormulae fs)

syntMatUsedInVectForm vf =
    case vf of
      vt1 :<===: vt2  -> joinSyntMat (syntMatUsedInVectTerm vt1)
                                     (syntMatUsedInVectTerm vt2)
      vt1 :>===: vt2  -> joinSyntMat (syntMatUsedInVectTerm vt1)
                                     (syntMatUsedInVectTerm vt2)
      vt1 :====: vt2  -> joinSyntMat (syntMatUsedInVectTerm vt1)
                                     (syntMatUsedInVectTerm vt2)
      vt1 :<=/=: vt2  -> joinSyntMat (syntMatUsedInVectTerm vt1)
                                     (syntMatUsedInVectTerm vt2)
      vt1 :>=/=: vt2  -> joinSyntMat (syntMatUsedInVectTerm vt1)
                                     (syntMatUsedInVectTerm vt2)
      vt1 :==/=: vt2  -> joinSyntMat (syntMatUsedInVectTerm vt1)
                                     (syntMatUsedInVectTerm vt2)
      VE  vt et       -> joinSyntMat (syntMatUsedInVectTerm vt)
                                     (syntMatUsedInElemTerm et)
      VectInSet vt vs -> joinSyntMat (syntMatUsedInVectTerm vt)
                                     (syntMatUsedInVectSet  vs)
      QuantVectForm _ vv fs -> joinSyntMat (syntMatUsedInVectVari vv)
                                           (syntMatUsedInFormulae fs)


syntMatUsedInRelaForm rf =
    case rf of
      rt1 :<==: rt2   -> joinSyntMat (syntMatUsedInRelaTerm rt1)
                                     (syntMatUsedInRelaTerm rt2)
      rt1 :>==: rt2   -> joinSyntMat (syntMatUsedInRelaTerm rt1)
                                     (syntMatUsedInRelaTerm rt2)
      rt1 :===: rt2   -> joinSyntMat (syntMatUsedInRelaTerm rt1)
                                     (syntMatUsedInRelaTerm rt2)
      rt1 :<=/: rt2   -> joinSyntMat (syntMatUsedInRelaTerm rt1)
                                     (syntMatUsedInRelaTerm rt2)
      rt1 :>=/: rt2   -> joinSyntMat (syntMatUsedInRelaTerm rt1)
                                     (syntMatUsedInRelaTerm rt2)
      rt1 :=/=: rt2   -> joinSyntMat (syntMatUsedInRelaTerm rt1)
                                     (syntMatUsedInRelaTerm rt2)
      REE  rt et1 et2 -> joinSyntMatSet [syntMatUsedInRelaTerm rt,
                                         syntMatUsedInElemTerm et1,
                                         syntMatUsedInElemTerm et2]
```

```
      RelaInSet rt rs -> joinSyntMat (syntMatUsedInRelaTerm rt)
                                     (syntMatUsedInRelaSet  rs)
      QuantRelaForm _ rv fs -> joinSyntMat (syntMatUsedInRelaVari rv)
                                           (syntMatUsedInFormulae fs)
syntMatUsedInRelaForms rfs = joinSyntMatSet
                                      (map syntMatUsedInRelaForm rfs)

syntMatUsedInPartForm rf =
    case rf of
      rt1 :<====: rt2 -> joinSyntMat (syntMatUsedInPartTerm rt1)
                                     (syntMatUsedInPartTerm rt2)
      rt1 :>====: rt2 -> joinSyntMat (syntMatUsedInPartTerm rt1)
                                     (syntMatUsedInPartTerm rt2)
      rt1 :<=/==: rt2 -> joinSyntMat (syntMatUsedInPartTerm rt1)
                                     (syntMatUsedInPartTerm rt2)
      rt1 :>=/==: rt2 -> joinSyntMat (syntMatUsedInPartTerm rt1)
                                     (syntMatUsedInPartTerm rt2)
      StrictPartContained pt1 pt2 ->
                          joinSyntMat (syntMatUsedInPartTerm pt1)
                                     (syntMatUsedInPartTerm pt2)


syntMatUsedInFormula f =
   case f of
      FV fv          -> syntMatUsedInFormVari fv
      EF ef          -> syntMatUsedInElemForm ef
      VF vf          -> syntMatUsedInVectForm vf
      RF rf          -> syntMatUsedInRelaForm rf
      --PF pf          -> syntMatUsedInPartForm pf
      Negated  f1    -> syntMatUsedInFormula  f1
      Implies  f1 f2 -> joinSyntMat (syntMatUsedInFormula f1)
                                    (syntMatUsedInFormula f2)
      SemEqu   f1 f2 -> joinSyntMat (syntMatUsedInFormula f1)
                                    (syntMatUsedInFormula f2)
      Disjunct f1 f2 -> joinSyntMat (syntMatUsedInFormula f1)
                                    (syntMatUsedInFormula f2)
      Conjunct f1 f2 -> joinSyntMat (syntMatUsedInFormula f1)
                                    (syntMatUsedInFormula f2)
syntMatUsedInFormulae fs = joinSyntMatSet (map syntMatUsedInFormula fs)
```

### 2.4.4 Inductively Determining Free Variables

Free variables may be determined. Here, they originate from individual variables which are used to define category objects, elements, vectors, and relations.

```
combineFreeVars fv1 fv2 =
   let (a,b,c,d) = fv1
       (u,v,w,x) = fv2
```

```
       plusNub x y = nub $ x ++ y
   in  (plusNub a u, plusNub b v, plusNub c w, plusNub d x)

combineFreeVarSets []    = ([],[],[],[])
combineFreeVarSets (h:t) = combineFreeVars h (combineFreeVarSets t)

freeVarInCatObj :: CatObject -> ([CatObjVar],[ElemVari],[VectVari],[RelaVari])
freeVarInCatObj o =
   case o of
      OV vo -> ([vo],[],[],[])
      _     -> ([  ],[],[],[])


freeVarInCatObjs :: [CatObject] -> ([CatObjVar],[ElemVari],[VectVari],[RelaVari])
freeVarInCatObjs cos = combineFreeVarSets (map freeVarInCatObj cos)

freeVarInParObj :: ParObject -> ([CatObjVar],[ElemVari],[VectVari],[RelaVari])
freeVarInParObj po =
   case po of
      ParObj co   -> freeVarInCatObj co
      ParPro o o' -> combineFreeVars (freeVarInParObj o )
                                     (freeVarInParObj o')
      ParSum o o' -> combineFreeVars (freeVarInParObj o )
                                     (freeVarInParObj o')
      ParPow o    -> freeVarInParObj o

freeVarInElemCnst :: ElemConst -> ([CatObjVar],[ElemVari],[VectVari],[RelaVari])
freeVarInElemCnst (Elem _ o) = freeVarInCatObj o

freeVarInElemVari :: ElemVari -> ([CatObjVar],[ElemVari],[VectVari],[RelaVari])
freeVarInElemVari ev =
   case ev of
      VarE          _ o -> combineFreeVars ([],[ev],[],[]) (freeVarInCatObj o)
      IndexedVarE _ _ o -> combineFreeVars ([],[ev],[],[]) (freeVarInCatObj o)

freeVarInElemTerm :: ElemTerm -> ([CatObjVar],[ElemVari],[VectVari],[RelaVari])
freeVarInElemTerm t =
   case t of
      EC    ec -> freeVarInElemCnst ec
      EV    ev -> freeVarInElemVari ev
      Pair et1 et2 -> combineFreeVars (freeVarInElemTerm et1)
                                      (freeVarInElemTerm et2)
      Inj1 et o    -> combineFreeVars (freeVarInElemTerm et)
                                      (freeVarInCatObj   o)
      Inj2 o  et   -> combineFreeVars (freeVarInElemTerm et)
                                      (freeVarInCatObj   o)
      ThatV vt -> freeVarInVectTerm vt
      SomeV vt -> freeVarInVectTerm vt
      ThatR rt -> freeVarInRelaTerm rt
      SomeR rt -> freeVarInRelaTerm rt
```

```
      FuncAppl fc et -> freeVarInElemTerm et
      VectToElem _ -> freeVarInElemTerm $ expandDefinedElemTerm t

freeVarInVectConst :: VectConst -> ([CatObjVar],[ElemVari],[VectVari],[RelaVari])
freeVarInVectConst (Vect _ o) = freeVarInCatObj o

freeVarInVectVari :: VectVari -> ([CatObjVar],[ElemVari],[VectVari],[RelaVari])
freeVarInVectVari vv =
   case vv of
      VarV          _ o -> combineFreeVars ([],[],[vv],[]) (freeVarInCatObj o)
      IndexedVarV _ _ o -> combineFreeVars ([],[],[vv],[]) (freeVarInCatObj o)

freeVarInVectTerm :: VectTerm -> ([CatObjVar],[ElemVari],[VectVari],[RelaVari])
freeVarInVectTerm vt =
   case vt of
      VC vc           -> freeVarInVectConst vc
      VV vv           -> freeVarInVectVari  vv
      rt  :****: vt2  -> combineFreeVars (freeVarInRelaTerm rt)
                                         (freeVarInVectTerm vt2)
      vt1 :||||: vt2  -> combineFreeVars (freeVarInVectTerm vt1)
                                         (freeVarInVectTerm vt2)
      vt1 :&&&&: vt2  -> combineFreeVars (freeVarInVectTerm vt1)
                                         (freeVarInVectTerm vt2)
      Syq rt  vt2     -> combineFreeVars (freeVarInRelaTerm rt)
                                         (freeVarInVectTerm vt2)
      NegaV      vt1  -> freeVarInVectTerm vt1
      NullV      o    -> freeVarInCatObj   o
      UnivV      o    -> freeVarInCatObj   o
      SupVect    vs   -> freeVarInVectSet  vs
      InfVect    vs   -> freeVarInVectSet  vs
      RelaToVect _    -> freeVarInVectTerm $ expandDefinedVectTerm vt
      PointVect  et   -> freeVarInElemTerm et
      VFctAppl vf vt2 -> combineFreeVars (freeVarInVectFct  vf)
                                         (freeVarInVectTerm vt2)
      PowElemToVect et -> freeVarInElemTerm et


freeVarInRelaConst :: RelaConst -> ([CatObjVar],[ElemVari],[VectVari],[RelaVari])
freeVarInRelaConst (Rela _ o o') = freeVarInCatObjs [o,o']

freeVarInFuncConst :: FuncConst -> ([CatObjVar],[ElemVari],[VectVari],[RelaVari])
freeVarInFuncConst (Func _ o o') = freeVarInCatObjs [o,o']

freeVarInRelaVari :: RelaVari -> ([CatObjVar],[ElemVari],[VectVari],[RelaVari])
freeVarInRelaVari rv =
   case rv of
      VarR          _ o o' -> combineFreeVars ([],[],[],[rv])
                                              (freeVarInCatObjs [o,o'])
      IndexedVarR _ _ o o' -> combineFreeVars ([],[],[],[rv])
                                              (freeVarInCatObjs [o,o'])
```

```
freeVarInRelaTerm :: RelaTerm -> ([CatObjVar],[ElemVari],[VectVari],[RelaVari])
freeVarInRelaTerm rt =
   case rt of
      RC rc          -> freeVarInRelaConst rc
      RV rv          -> freeVarInRelaVari  rv
      rt1 :***: rt2  -> combineFreeVars (freeVarInRelaTerm rt1)
                                        (freeVarInRelaTerm rt2)
      rt1 :|||: rt2  -> combineFreeVars (freeVarInRelaTerm rt1)
                                        (freeVarInRelaTerm rt2)
      rt1 :&&&: rt2  -> combineFreeVars (freeVarInRelaTerm rt1)
                                        (freeVarInRelaTerm rt2)
      NegaR     rt1  -> freeVarInRelaTerm rt1
      Ident      d   -> freeVarInCatObj d
      NullR      d c -> freeVarInCatObjs [c,d]
      UnivR      d c -> freeVarInCatObjs [c,d]
      Convs      rt1 -> freeVarInRelaTerm rt1
      vt1 :||--: vt2 -> combineFreeVars (freeVarInVectTerm vt1)
                                        (freeVarInVectTerm vt2)
      SupRela    rs  -> freeVarInRelaSet  rs
      InfRela    rs  -> freeVarInRelaSet  rs
      (:*:)    _ _   -> freeVarInRelaTerm $ expandDefinedRelaTerm rt
      (:\/:)   _ _   -> freeVarInRelaTerm $ expandDefinedRelaTerm rt
      Pi       o o'  -> combineFreeVars (freeVarInCatObj o )
                                        (freeVarInCatObj o')
      Rho      o o'  -> combineFreeVars (freeVarInCatObj o )
                                        (freeVarInCatObj o')
      Iota     o o'  -> combineFreeVars (freeVarInCatObj o )
                                        (freeVarInCatObj o')
      Kappa    o o'  -> combineFreeVars (freeVarInCatObj o )
                                        (freeVarInCatObj o')
      CASE rt1 rt2   -> combineFreeVars (freeVarInRelaTerm rt1)
                                        (freeVarInRelaTerm rt2)
      Project  rt1   -> freeVarInRelaTerm rt1
      Epsi     o     -> freeVarInCatObj   o
      PointDiag  et  -> freeVarInElemTerm et
      SyQ      _ _   -> freeVarInRelaTerm $ expandDefinedRelaTerm rt
      --Wait       rt1 -> freeVarInPartTerm rt1
      Belly pt         -> freeVarInPartTerm pt
      InjTerm    vt  -> freeVarInVectTerm vt
      ProdVectToRela _ -> freeVarInRelaTerm $ expandDefinedRelaTerm rt
      PartOrd po       -> freeVarInParObj po
      RFctAppl rf rt2 -> combineFreeVars (freeVarInRelaFct rf)
                                         (freeVarInEVRTerm rt2)
freeVarInEVRTerm evr =
   case evr of
      ArgE ae -> freeVarInElemTerm ae
      ArgV av -> freeVarInVectTerm av
      ArgR ar -> freeVarInRelaTerm ar
```

```
freeVarInPartTerm :: PartTerm -> ([CatObjVar],[ElemVari],[VectVari],[RelaVari])
freeVarInPartTerm pt =
   case pt of
      Lift   rt1      -> freeVarInRelaTerm rt1
      Fetus po1 rt1 po2 -> combineFreeVarSets [freeVarInParObj   po1,
                                               freeVarInRelaTerm rt1,
                                               freeVarInParObj   po2]
      pt1 :*****: pt2 -> combineFreeVars (freeVarInPartTerm pt1)
                                         (freeVarInPartTerm pt2)
      pt1 :|||||: pt2 -> combineFreeVars (freeVarInPartTerm pt1)
                                         (freeVarInPartTerm pt2)
      pt1 :&&&&&: pt2 -> combineFreeVars (freeVarInPartTerm pt1)
                                         (freeVarInPartTerm pt2)
      NegaPart pt1    -> freeVarInPartTerm pt1
      IdentP po       -> freeVarInParObj po
      NullP  d c      -> combineFreeVars (freeVarInParObj d)
                                         (freeVarInParObj c)
      UnivP  d c      -> combineFreeVars (freeVarInParObj d)
                                         (freeVarInParObj c)
      TranspP pt1     -> freeVarInPartTerm pt1
      PPi        o o' -> combineFreeVars (freeVarInParObj o )
                                         (freeVarInParObj o')
      PRho       o o' -> combineFreeVars (freeVarInParObj o )
                                         (freeVarInParObj o')
      pt1 :#: pt2     -> combineFreeVars (freeVarInPartTerm pt1)
                                         (freeVarInPartTerm pt2)
      pt1 :\\//: pt2  -> combineFreeVars (freeVarInPartTerm pt1)
                                         (freeVarInPartTerm pt2)
      PIota      o o' -> combineFreeVars (freeVarInParObj o )
                                    (freeVarInParObj o')
      PKappa     o o' -> combineFreeVars (freeVarInParObj o )
                                    (freeVarInParObj o')
      PEpsi      o    -> freeVarInParObj o


freeVarInVectFct :: VectFct -> ([CatObjVar],[ElemVari],[VectVari],[RelaVari])
freeVarInVectFct (VFCT vv vt) = (a, b,filter (/= vv) c, d)
                                      where (a,b,c,d) = freeVarInVectTerm vt
freeVarInRelaFct :: RelaFct -> ([CatObjVar],[ElemVari],[VectVari],[RelaVari])
freeVarInRelaFct (RFCT evr rt) =
   case evr of
      EVar ea -> (a, filter (/= ea) b, c, d) where (a,b,c,d) = freeVarInRelaTerm rt
      VVar va -> (a, b, filter (/= va) c, d) where (a,b,c,d) = freeVarInRelaTerm rt
      RVar ra -> (a, b, c, filter (/= ra) d) where (a,b,c,d) = freeVarInRelaTerm rt

freeVarInElemSet :: ElemSET -> ([CatObjVar],[ElemVari],[VectVari],[RelaVari])
freeVarInElemSet es =
   case es of
      VarES _ o -> freeVarInCatObj o
      ET    o   -> freeVarInCatObj o
```

```
      ES ev fs  -> let (a,b,c,d) = freeVarInFormulae fs
                   in  (a,filter (/= ev) b,c,d)
      EX es o   -> combineFreeVarSets ((map freeVarInElemTerm es) ++
                                       [freeVarInCatObj o])
freeVarInVectSet :: VectSET -> ([CatObjVar],[ElemVari],[VectVari],[RelaVari])
freeVarInVectSet vs =
   case vs of
      VarVS _ o -> freeVarInCatObj o
      VS vv fs  -> let (a,b,c,d) = freeVarInFormulae fs
                   in  (a,b,filter (/= vv) c,d)
      VX vs o   -> combineFreeVarSets ((map freeVarInVectTerm vs) ++
                                       [freeVarInCatObj o])


freeVarInRelaSet :: RelaSET -> ([CatObjVar],[ElemVari],[VectVari],[RelaVari])
freeVarInRelaSet rs =
   case rs of
      VarRS _ o o' -> freeVarInCatObjs [o,o']
      RS rv fs     -> let (a,b,c,d) = freeVarInFormulae fs
                      in  (a,b,c,filter (/= rv) d)
      RX rs o o'   -> combineFreeVarSets ((map freeVarInRelaTerm rs) ++
                                          [freeVarInCatObjs [o,o']])

freeVarInElemForm :: ElemForm -> ([CatObjVar],[ElemVari],[VectVari],[RelaVari])
freeVarInElemForm f =
   case f of
      Equation t1 t2 -> combineFreeVars (freeVarInElemTerm t1)
                                        (freeVarInElemTerm t2)
      NegaEqua t1 t2 -> combineFreeVars (freeVarInElemTerm t1)
                                        (freeVarInElemTerm t2)
      QuantElemForm _ ev fs -> (a,filter (/= ev) b, c, d)
                                         where (a,b,c,d) = freeVarInFormulae fs

freeVarInVectForm :: VectForm -> ([CatObjVar],[ElemVari],[VectVari],[RelaVari])
freeVarInVectForm vf =
  case vf of
      vt1 :<===: vt2          -> combineFreeVars (freeVarInVectTerm vt1)
                                                 (freeVarInVectTerm vt2)
      vt1 :>===: vt2          -> combineFreeVars (freeVarInVectTerm vt1)
                                                 (freeVarInVectTerm vt2)
      vt1 :====: vt2          -> combineFreeVars (freeVarInVectTerm vt1)
                                                 (freeVarInVectTerm vt2)
      vt1 :<=/=: vt2          -> combineFreeVars (freeVarInVectTerm vt1)
                                                 (freeVarInVectTerm vt2)
      vt1 :>=/=: vt2          -> combineFreeVars (freeVarInVectTerm vt1)
                                                 (freeVarInVectTerm vt2)
      vt1 :==/=: vt2          -> combineFreeVars (freeVarInVectTerm vt1)
                                                 (freeVarInVectTerm vt2)
      VectInSet vt vs         -> combineFreeVars (freeVarInVectTerm vt)
                                                 (freeVarInVectSet  vs)
```

```
      VE     vt t              -> combineFreeVars (freeVarInElemTerm t)
                                                  (freeVarInVectTerm vt)
      QuantVectForm _ vv fs -> (a,b,filter (/= vv) c,d)
                                        where (a,b,c,d) = freeVarInFormulae fs


freeVarInRelaForm :: RelaForm -> ([CatObjVar],[ElemVari],[VectVari],[RelaVari])
freeVarInRelaForm rf =
  case rf of
      rt1 :<==: rt2           -> combineFreeVars (freeVarInRelaTerm rt1)
                                                 (freeVarInRelaTerm rt2)
      rt1 :>==: rt2           -> combineFreeVars (freeVarInRelaTerm rt1)
                                                 (freeVarInRelaTerm rt2)
      rt1 :===: rt2           -> combineFreeVars (freeVarInRelaTerm rt1)
                                                 (freeVarInRelaTerm rt2)
      rt1 :<=/: rt2           -> combineFreeVars (freeVarInRelaTerm rt1)
                                                 (freeVarInRelaTerm rt2)
      rt1 :>=/: rt2           -> combineFreeVars (freeVarInRelaTerm rt1)
                                                 (freeVarInRelaTerm rt2)
      rt1 :=/=: rt2           -> combineFreeVars (freeVarInRelaTerm rt1)
                                                 (freeVarInRelaTerm rt2)
      REE             rt t1 t2 -> combineFreeVarSets [freeVarInElemTerm t1,
                                                      freeVarInElemTerm t2,
                                                      freeVarInRelaTerm rt]
      RelaInSet          rt rs -> combineFreeVars (freeVarInRelaTerm rt)
                                                  (freeVarInRelaSet  rs)
      QuantRelaForm _ rv fs -> (a,b,c,filter (/= rv) d)
                                        where (a,b,c,d) = freeVarInFormulae fs
freeVarInRelaForms :: [RelaForm] -> ([CatObjVar],[ElemVari],[VectVari],[RelaVari])
freeVarInRelaForms rfs = combineFreeVarSets (map freeVarInRelaForm rfs)

freeVarInPartForm :: PartForm -> ([CatObjVar],[ElemVari],[VectVari],[RelaVari])
freeVarInPartForm rf =
  case rf of
      rt1 :<====: rt2 -> combineFreeVars (freeVarInPartTerm rt1)
                                         (freeVarInPartTerm rt2)
      rt1 :>====: rt2 -> combineFreeVars (freeVarInPartTerm rt1)
                                         (freeVarInPartTerm rt2)
      rt1 :<=/==: rt2 -> combineFreeVars (freeVarInPartTerm rt1)
                                         (freeVarInPartTerm rt2)
      rt1 :>=/==: rt2 -> combineFreeVars (freeVarInPartTerm rt1)
                                         (freeVarInPartTerm rt2)
      StrictPartContained pt1 pt2 -> combineFreeVars
                                         (freeVarInPartTerm pt1)
                                         (freeVarInPartTerm pt2)


freeVarInFormVari :: FormVari -> ([CatObjVar],[ElemVari],[VectVari],[RelaVari])
freeVarInFormVari fv = ([],[],[],[])

freeVarInFormula :: Formula -> ([CatObjVar],[ElemVari],[VectVari],[RelaVari])
```

```
freeVarInFormula f =
   case f of
      EF ef           -> freeVarInElemForm ef
      VF vf           -> freeVarInVectForm vf
      RF rf           -> freeVarInRelaForm rf
      --PF pf          -> freeVarInPartForm pf
      Verum           -> ([],[],[],[])
      Falsum          -> ([],[],[],[])
      Negated   f1    -> freeVarInFormula f1
      Implies   f1 f2 -> combineFreeVars (freeVarInFormula f1)
                                         (freeVarInFormula f2)
      SemEqu    f1 f2 -> combineFreeVars (freeVarInFormula f1)
                                         (freeVarInFormula f2)
      Conjunct  f1 f2 -> combineFreeVars (freeVarInFormula f1)
                                         (freeVarInFormula f2)
      Disjunct  f1 f2 -> combineFreeVars (freeVarInFormula f1)
                                         (freeVarInFormula f2)
freeVarInFormulae :: [Formula] -> ([CatObjVar],[ElemVari],[VectVari],[RelaVari])
freeVarInFormulae fs = combineFreeVarSets (map freeVarInFormula fs)

isClosedElemForm ef = freeVarInElemForm ef == ([],[],[],[])
isClosedVectForm vf = freeVarInVectForm vf == ([],[],[],[])
isClosedRelaForm rf = freeVarInRelaForm rf == ([],[],[],[])
isClosedFormula  f  = freeVarInFormula  f  == ([],[],[],[])
```

## 2.5 Theories

We now 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
                 [FuncConst]    -- function denotations encountered in the theory
                 [VectFct]      -- vector functions encountered in the theory
                 [RelaFct]      -- relation functions encountered in the theory
                 [Formula]      -- formulae demanded to hold
     deriving (Read, Show, Eq, Ord)


checkTheoryWellDefined th =
   let TH s os cs vs rs fs _ _ f = th         -- functions still unhandled
       objectsInElemConsts = map domEC cs
       objectsInVectConsts = map domVC vs
       objectsInRelaConsts = concatMap (\(Rela _ so to) -> [so, to]) rs
       objectsInFuncConsts = concatMap (\(Func _ so to) -> [so, to]) fs
       eightItems@(l,m,_,_,_,_,_,_,_,_) = syntMatUsedInFormulae f
       objNonGeneric o =
```

```
          case o of
             DirPro _ _ -> False
             DirSum _ _ -> False
             DirPow _   -> False
             UnitOb     -> False
             QuotMod  _ -> False
             InjFrom  _ -> False
             _          -> True
       nonGenericObjects = filter objNonGeneric $
            objectsInElemConsts ++ objectsInVectConsts ++ objectsInRelaConsts ++
            (map OV l) ++ (map OC m)
       onlySyntMatFromTheory (_,_,_,d,_,f,_,h,i,_) =
          let constsOK = and (map (`elem` cs) d)
              vectsOK  = and (map (`elem` vs) f)
              relsOK   = and (map (`elem` rs) h)
              fctsOK   = and (map (`elem` fs) i)
          in  constsOK && vectsOK && relsOK && fctsOK
   in  (sort os == (sort $ nub nonGenericObjects)) &&
       (namesDisjointT th) && (onlySyntMatFromTheory eightItems)


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

One will often wish to check whether a term is built from the basics of some theory. The following functions checks this, not yet, however, for vector or relation functions. Also, generic constructs are not checked for as they are considered to be available all the time.

```
relaTermIsBuiltFromTheory rt th =
   let (sMov,sMoc,sMev,sMec,sMvv,sMvc,sMrv,sMrc,sMfc,_) = syntMatUsedInRelaTerm rt
       TH _ ocs ecs vcs rcs fcs vfs rfs fs = th
       objectsUsed = map OV sMov ++ map OC sMoc
       objectsContained    = map (`elem` ocs) objectsUsed
       elemConstsContained = map (`elem` ecs) sMec
```

```
      vectConstsContained = map (`elem` vcs) sMvc
      relaConstsContained = map (`elem` rcs) sMrc
      funcConstsContained = map (`elem` fcs) sMfc
   in  and $
      objectsContained    ++ elemConstsContained ++
      relaConstsContained ++ funcConstsContained


whyNotRelaTermIsBuiltFromTheory rt th =
   let (sMov,sMoc,sMev,sMec,sMvv,sMvc,sMrv,sMrc,sMfc,_) = syntMatUsedInRelaTerm rt
      TH _ ocs ecs vcs rcs fcs vfs rfs fs = th
      objectsUsed = map OV sMov ++ map OC sMoc
      objectsContained    = map (`elem` ocs) objectsUsed
      nonContainedObjects = case and objectsContained of
                              True  -> []
                              False -> filter (`notElem` ocs) objectsUsed
      elemConstsContained = map (`elem` ecs) sMec
      nonContainedElements  = case and elemConstsContained of
                              True  -> []
                              False -> filter (`notElem` ecs) sMec
      vectConstsContained = map (`elem` vcs) sMvc
      nonContainedVectors = case and vectConstsContained of
                              True  -> []
                              False -> filter (`notElem` vcs) sMvc
      relaConstsContained   = map (`elem` rcs) sMrc
      nonContainedRelations = case and relaConstsContained of
                                True  -> []
                                False -> filter (`notElem` rcs) sMrc
      funcConstsContained   = map (`elem` fcs) sMfc
      nonContainedFunctions = case and funcConstsContained of
                                True  -> []
                                False -> filter (`notElem` fcs) sMfc
   in  map show nonContainedObjects   ++
      map show nonContainedElements  ++
      map show nonContainedVectors   ++
      map show nonContainedRelations ++
      map show nonContainedFunctions
```

# 3 Semantics

While we have so far only been concerned with syntax, we will now offer the opportunity to interpret the language, and the theories we have defined, in a model. A basic distinction will be made between "normal" relations and partialities. The former may be formulated starting from constants and variables as usual, and the constants need an interpretation. For the latter no constants are provided, and consequently no interpretation is necessary. An interpretation for partialities will only be possible along the generic methods of formulating them.

## 3.1 Models

Via an interpretation, the objects get assigned sets in the 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, a boolean vector or matrix respectively. The element constant gets assigned the number of the entry, i.e., an integer.

```
data InterpretObjs = Carrier  CatObject   Int     deriving (Eq, Show, Read, Ord)
data InterpretCons = InterCon ElemConst   Int     deriving (Eq, Show, Read, Ord)
data InterpretVect = InterVec VectConst  [Bool]   deriving (Eq, Show, Read, Ord)
data InterpretRela = InterRel RelaConst [[Bool]]  deriving (Eq, Show, Read, Ord)
data InterpretFunc = InterFct FuncConst  [Int]    deriving (Eq, Show, Read, Ord)
data InterpretVFct = InterVFc VectFct   ([Bool]  ->  [Bool])
data EVRArg = EArg Int | VArg  [Bool] | RArg [[Bool]]
                                                  deriving (Eq, Ord, Read, Show)
data InterpretRFct = InterRFc RelaFct  (EVRArg -> [[Bool]])
instance Show InterpretVFct where
   show (InterVFc vf vt) = show "VectorFunctions"
instance Show InterpretRFct where
   show (InterRFc rf rt) = show "RelationFunctions"
```

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

```
data Model = MO String            -- name of the model
               Theory             -- underlying theory of the model
               [InterpretObjs]    -- cardinalities of carrier sets
               [InterpretCons]    -- numbers of corresponding elements
               [InterpretVect]    -- subset-interpreting boolean vectors
               [InterpretRela]    -- relation-interpreting matrices
```

```
              [InterpretFunc]   -- function-interpreting functions
              [InterpretVFct]   -- interpreted vector functions
              [InterpretRFct] | -- interpreted relation functions
            RATHModel String  |
            KUREModel String               deriving (Show)
```

The additional two variants are just indications where to embed possible future models extending the present report. All the interpreting functions should then respect these variants by introducing the respective case analyses.

We provide some mechanisms on the model side to check, whether the sets in question are assigned to objects consistently by the interpretations.

```
getElemConstInterpretation cs c =
   (\(InterCon _ b) -> b) $ head $ dropWhile (\(InterCon e _) -> c /= e) cs
getVectConstInterpretation vs v =
   (\(InterVec _ b) -> b) $ head $ dropWhile (\(InterVec e _) -> v /= e) vs
getRelaConstInterpretation rs r =
   (\(InterRel _ b) -> b) $ head $ dropWhile (\(InterRel e _) -> r /= e) rs
getFuncConstInterpretation fs f =
   (\(InterFct _ b) -> b) $ head $ dropWhile (\(InterFct e _) -> f /= e) fs



arities (MO _ _ os _ _ _ _ _ _) = map (\(Carrier o n) -> (o, n)) os
arity m env o =
   case o of
      -- OV ov -> should not occur in an interpretation!
      OC oc -> snd (head $ dropWhile (\(o',_) -> o' /= o) (arities m))
      DirPro o1 o2 -> arity m env o1 * (arity m env o2)
      DirSum o1 o2 -> arity m env o1 + (arity m env o2)
      DirPow o1    -> 2^(arity m env o1)
      UnitOb       -> 1
      QuotMod rt -> let equRel = interpretRelaTerm m env rt
                    in  length $ nub equRel
      InjFrom vt -> let subSet = interpretVectTerm m env vt
                    in  length subSet
      -- Strict  po ->
```

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

```
namesDisjointM (MO s th os cs vs rs fs _ _) =
    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
        funcNames = nub $ map (\(InterFct (Func s _ _) _)  -> s) fs
        lcs = length cs
        lvs = length vs
        lrs = length rs
        lfs = length fs
        cDisjoint = length elemNames == lcs
```

```
        vDisjoint = length vectNames == lvs
        rDisjoint = length relaNames == lrs
        fDisjoint = length funcNames == lfs
        allDisjoint = lcs + lvs + lrs + lfs ==
           length (nub (elemNames ++ vectNames ++ relaNames ++ funcNames))
    in  cDisjoint  && vDisjoint && rDisjoint   && fDisjoint               &&
        (s `notElem` elemNames) && allDisjoint && (s `notElem` vectNames) &&
        (s `notElem` relaNames) && (s `notElem` funcNames) && namesDisjointT th

aritiesConsistent mo@(MO s _ os cs vs rs fs vfs rfs) =
    let ooo (Carrier          o  n)    = (o, n)
        eee (InterCon (Elem _ o) n)    = (o, n)
        vvv (InterVec (Vect _ o) v)    = (o, length v)
        rrr (InterRel (Rela _ o o') m) = [(o, rows m), (o', cols m)]
        fff (InterFct (Func _ o _ ) f) = (o, length f)
        ff1 (InterFct (Func _ _ o') f) = map (\x -> (o',x)) f
        ttt (InterVFc (VFCT  vv vt) m) = [(dvv,arity mo ([],[],[]) dvv),
                                          (dvt,arity mo ([],[],[])  dvt)]
                                            where dvv = domVV vv
                                                  dvt = domVT vt
        uuu (InterRFc (RFCT  evr rt) m) =
           case evr of
              EVar ea -> [(drv,arity mo ([],[],[])  drv),
                                          (drt,arity mo ([],[],[])  drt),
                                          (crt,arity mo ([],[],[])  crt)]
                                            where drv = domEV ea
                                                  drt = domRT rt
                                                  crt = codRT rt
              VVar va -> [(drv,arity mo ([],[],[])  drv),
                                          (drt,arity mo ([],[],[])  drt),
                                          (crt,arity mo ([],[],[])  crt)]
                                            where drv = domVV va
                                                  drt = domRT rt
                                                  crt = codRT rt
              RVar ra -> [(drv,arity mo ([],[],[])  drv),
                                          (drt,arity mo ([],[],[])  drt),
                                          (crv,arity mo ([],[],[])  crv),
                                          (crt,arity mo ([],[],[])  crt)]
                                            where drv = domRV ra
                                                  crv = codRV ra
                                                  drt = domRT rt
                                                  crt = codRT rt
        carrSetAndSize  = map ooo os
        elemNumbInSets  = map eee cs
        vectSetAndSize  = map vvv vs
        relaSetAndSize  = concatMap rrr rs
        funcSetAndSize  = map fff fs
        vectFctAndSize  = concatMap ttt vfs
        relaFctAndSize  = concatMap uuu rfs
        noDiscrepancies = length carrSetAndSize ==
```

```
            (length $ nub (carrSetAndSize ++ vectSetAndSize ++ relaSetAndSize ++
                           funcSetAndSize ++ vectFctAndSize ++ relaFctAndSize))
        elementIsInSet (o,e) =
            e <= (snd $ head $ dropWhile (\x -> o /= fst x) carrSetAndSize)
        allElementsInside   = and (map elementIsInSet
                                       (elemNumbInSets ++ concatMap ff1 fs))
    in  noDiscrepancies && allElementsInside
```

## 3.2 Interpretation

Several forms of an interpretation are aimed at. We hope to offer an interpretation in RelView or in Rath. But sometimes one will have a small problem that shall be investigated quickly. In these cases, the following interpretation may be taken. Before the interpretation is possible, we need valuations of the individual variables. Using a rather primitive lookup version, the rest is then standard.

```
type ValuateElemVari = (ElemVari,   Int)
type ValuateVectVari = (VectVari,  [Bool])
type ValuateRelaVari = (RelaVari, [[Bool]])


type ElemValuations     = [ValuateElemVari]
type VectValuations     = [ValuateVectVari]
type RelaValuations     = [ValuateRelaVari]
type Environment        = (ElemValuations,VectValuations,RelaValuations)


lookupPrimitive :: Eq a => a -> [(a,b)] -> b
lookupPrimitive k [] = error "empty lookup list"
lookupPrimitive k ((x,y):xys)
      | k == x    = y
      | otherwise = lookupPrimitive k xys

valuation env v = lookupPrimitive v env
```

Now we can start interpreting items of the language.

```
interpretElemConst :: Model -> Environment -> ElemConst -> Int
interpretElemConst m env ec =
   case ec of
      Elem _ _ -> let MO _ _ _ es _ _ _ _ _ = m
                  in  getElemConstInterpretation es ec
      GenericElemNotation o i ->
         let elemNumber = arity m env o
         in  case i <= elemNumber of
                True  -> i
                False -> error "error in generic element number"
```

```
interpretElemVari :: Environment -> ElemVari -> Int
interpretElemVari env v =
   let (ievs,_,_) = env
   in  valuation ievs v

interpretElemTerm :: Model -> Environment -> ElemTerm -> Int
interpretElemTerm m env t =
   case t of
      EC    ec -> interpretElemConst m env ec
      EV    ev -> interpretElemVari env ev
      Pair et1 et2 -> let et1Interpreted = interpretElemTerm m env et1
                          et2Interpreted = interpretElemTerm m env et2
                          elemNo2 = arity m env (domET et2)
                      in  ((et1Interpreted - 1) * elemNo2) + et2Interpreted
      Inj1 et _  -> interpretElemTerm m env et
      Inj2 o  et -> arity m env o + (interpretElemTerm m env et)
      ThatV vt -> let vtInterpreted = interpretVectTerm m env vt
                      zipped = zip [1..] vtInterpreted
                  in  fst $ head $ filter snd zipped
      SomeV vt -> let vtInterpreted = interpretVectTerm m env vt
                      zipped = zip [1..] vtInterpreted
                  in  fst $ head $ filter snd zipped   --taking the first
      ThatR rt -> let rtInterpreted = interpretRelaTerm m env rt
                      zipped = map (\(nr,line) -> (nr,line !! (nr-1))) $
                                   zip [1..] rtInterpreted
                  in  fst $ head $ filter snd zipped
      SomeR rt -> let rtInterpreted = interpretRelaTerm m env rt
                      zipped = map (\(nr,line) -> (nr,line !! (nr-1))) $
                                   zip [1..] rtInterpreted
                  in  fst $ head $ filter snd zipped   --taking the first
      FuncAppl fc et -> let etInterpreted = interpretElemTerm m env et
                            fcInterpreted = interpretFuncConst m fc
                        in  fcInterpreted !! etInterpreted
      VectToElem _ -> interpretElemTerm m env $ expandDefinedElemTerm t
      -- EFctAppl eFct et -> let ivf = interpretElemFct  m env eFct
      --                         ivt = interpretElemTerm m env et
      --                     in  ivf ivt


interpretVectConst :: Model -> VectConst -> [Bool]
interpretVectConst m vc =
   let MO _ _ _ _ vs _ _ _ _ = m
   in  getVectConstInterpretation vs vc
interpretVectVari :: Environment -> VectVari -> [Bool]
interpretVectVari env v =
   let (_,vvs,_) = env
   in  valuation vvs v
```

```
interpretVectTerm :: Model -> Environment -> VectTerm -> [Bool]
interpretVectTerm m env vt =
   let MO _ _ os _ vs _ _ _ _ = m
       (evs,vvs,rvs) = env
       impliesElem p q = not p || q
       impliesVect  = zipWith impliesElem
   in  case vt of
          VC vc           -> interpretVectConst m vc
          VV vv           -> interpretVectVari env vv
          rt1 :****: vt2  -> peirceProd   (interpretRelaTerm m env rt1)
                                          (interpretVectTerm m env vt2)
          vt1 :||||: vt2  -> zipWith (||) (interpretVectTerm m env vt1)
                                          (interpretVectTerm m env vt2)
          vt1 :&&&&: vt2  -> zipWith (&&) (interpretVectTerm m env vt1)
                                          (interpretVectTerm m env vt2)
          Syq       _ _   -> interpretVectTerm m env $ expandDefinedVectTerm vt
          NegaV       vt1 -> map not $ interpretVectTerm m env vt1
          NullV       o   -> replicate (arity m env o) False
          UnivV       o   -> replicate (arity m env o) True
          SupVect     vs  -> let VS vv fs = vs    --only if sup-hereditary
                                 size = arity m env $ domVV vv
                                 allVects = [ r | r <- transpMat (epsi size)]
                                 nextBigger currMax nextVector =
                                    if   and (impliesVect nextVector currMax)
                                    then currMax
                                    else case interpretFormulae
                                              m (evs,(vv,nextVector):vvs,rvs) fs of
                                           True  -> zipWith (||) nextVector
                                                                 currMax
                                           False -> currMax
                                 filtNextBigger currMax []    = currMax
                                 filtNextBigger currMax (h:t) = filtNextBigger nx t
                                         where nx = nextBigger currMax h
                             in  filtNextBigger (take size (repeat False)) allVects
          InfVect     vs  -> let VS vv fs = vs    --only if inf-hereditary
                                 size = arity m env $ domVV vv
                                 allVects = [ r | r <- transpMat (epsi size)]
                                 nextSmaller currMin nextVector =
                                    if   and (impliesVect nextVector currMin)
                                    then currMin
                                    else case interpretFormulae
                                              m (evs,(vv,nextVector):vvs,rvs) fs of
                                           True  -> zipWith (&&) nextVector
                                                                 currMin
                                           False -> currMin
                                 filtNextSmaller currMin []    = currMin
                                 filtNextSmaller currMin (h:t) = filtNextSmaller n t
                                         where n = nextSmaller currMin h
                             in  filtNextSmaller (take size (repeat False)) allVects
          RelaToVect    _ -> interpretVectTerm m env $ expandDefinedVectTerm vt
```

```
          PowElemToVect _ -> interpretVectTerm m env $ expandDefinedVectTerm vt
          PointVect  et   -> take (arity m env $ domET et)
                             (map (== interpretElemTerm m env et) [1..])
          VFctAppl vf vt2 -> let ivf = interpretVectFct  m env vf
                                 ivt = interpretVectTerm m env vt2
                             in  ivf ivt


interpretRelaConst :: Model -> RelaConst -> [[Bool]]
interpretRelaConst m rc =
   let MO _ _ _ _ _ rs _ _ _ = m
   in  getRelaConstInterpretation rs rc
interpretRelaVari :: Environment -> RelaVari -> [[Bool]]
interpretRelaVari env rv =
   let (_,_,rvs) = env
   in  valuation rvs rv
interpretFuncConst :: Model -> FuncConst -> [Int]
interpretFuncConst m fc =
   let MO _ _ _ _ _ _ fs _ _ = m
   in  getFuncConstInterpretation fs fc


interpretRelaTerm :: Model -> Environment -> RelaTerm -> [[Bool]]
interpretRelaTerm m env rt =
   let MO _ _ os _ _ rs _ _ _ = m
   in  case rt of
          RC rc          -> interpretRelaConst m rc
          RV rv          -> interpretRelaVari env rv
          rt1 :***: rt2  -> interpretRelaTerm m env rt1 ***
                            interpretRelaTerm m env rt2
          rt1 :|||: rt2  -> interpretRelaTerm m env rt1 |||
                            interpretRelaTerm m env rt2
          rt1 :&&&: rt2  -> interpretRelaTerm m env rt1 &&&
                            interpretRelaTerm m env rt2
          NegaR      rt1 -> negaMat $ interpretRelaTerm m env rt1
          Ident      d   -> ident (arity m env d)
          NullR      d c -> nulMatNM (arity m env d) (arity m env c)
          UnivR      d c -> allMatNM (arity m env d) (arity m env c)
          Convs      rt1 -> transpMat $ interpretRelaTerm m env rt1
          vt1 :||--: vt2 -> (map (\i -> [i]) (interpretVectTerm m env vt1)) ***
                            [interpretVectTerm m env vt2]
          SupRela rs -> let matrixSet = interpretRelaSET m env rs
                            (o,o')    = typeOfRS rs
                            nullRel   = nulMatNM (arity m env o) (arity m env o')
                        in  foldr (|||) nullRel matrixSet
          InfRela rs -> let matrixSet = interpretRelaSET m env rs
                            (o,o')    = typeOfRS rs
                            allRel    = allMatNM (arity m env o) (arity m env o')
                        in  foldr (&&&) allRel matrixSet
          Pi       o o'  -> let lengthO  = arity m env o
                                lengthO' = arity m env o'
```

```
                        rrooww i = concat (replicate (i-1) $
                                           replicate lengthO' False) ++
                                           replicate lengthO' True   ++
                                           concat (replicate (lengthO - i) $
                                           replicate lengthO' False)
                    in  transpMat $ map rrooww [1..lengthO]
Rho      o o'   -> let lengthO  = arity m env o
                       lengthO' = arity m env o'
                       identR   = ident lengthO'
                    in  concat $ replicate lengthO identR
(:*:)    _ _    -> interpretRelaTerm m env $ expandDefinedRelaTerm rt
(:\/:)   _ _    -> interpretRelaTerm m env $ expandDefinedRelaTerm rt
SyQ      _ _    -> interpretRelaTerm m env $ expandDefinedRelaTerm rt
Iota     o o'   -> let lengthO  = arity m env o
                       lengthO' = arity m env o'
                       identL   = ident lengthO
                       nullL    = nulMatNM lengthO' lengthO
                    in transpMat $ identL ++ nullL
Kappa    o o'   -> let lengthO  = arity m env o
                       lengthO' = arity m env o'
                       identR   = ident lengthO'
                       nullR    = nulMatNM lengthO lengthO'
                    in  transpMat $ nullR ++ identR
CASE rt1 rt2    -> interpretRelaTerm m env rt1 ++
                    interpretRelaTerm m env rt2
Project  rt1    -> let interpretedEquiv = interpretRelaTerm m env rt1
                       isEquiv = isEquivalence interpretedEquiv
                    in  case isEquiv of
                           True  -> transpMat $ nub interpretedEquiv
                           False -> error "is not an equivalence"
Epsi     o      -> epsi $ arity m env o
PointDiag et    -> pv *** (transpMat pv)
                         where pv = map (\i -> [i])
                                (interpretVectTerm m env (PointVect et))
InjTerm   vt    -> let vector = interpretVectTerm m env vt
                       fullIdentity = ident $ length vector
                    in  map fst $ filter snd $ zip fullIdentity vector
--Wait rt1       -> waiting (interpretPartTerm m env rt1)
Belly pt        -> g where BMGlobal _ _ g = convertBMToGlobal $
                                            interpretPartTerm m env pt
ProdVectToRela _ -> interpretRelaTerm m env $ expandDefinedRelaTerm rt
PartOrd po      -> expOrderings $ getObjectSupportStructure m env po
-- RFctAppl rf rt2  ->
--   let irf = interpretRelaFct m env rf
--       irt = case rt2 of
--               ArgE eval -> interpretElemTerm m env eval
--               ArgV vval -> interpretVectTerm m env vval
--               ArgR rval -> interpretRelaTerm m env rval
--   in  irf irt
RFctAppl rf evr  ->
```

```
            let (evs,vvs,rvs) = env
                RFCT evrVar rt1 = rf
            in  case evrVar of                -- strict evaluation
                   EVar ev ->
                      case evr of     -- may be better only substitution?
                         ArgE ae ->
                            interpretRelaTerm m
                               ((ev,interpretElemTerm m env ae):evs,vvs,rvs) rt1
                         _       -> error "wrong argument in RelaFct"
                   VVar vv ->
                      case evr of
                         ArgV av ->
                            interpretRelaTerm m
                               (evs,(vv,interpretVectTerm m env av):vvs,rvs) rt1
                         _       -> error "wrong argument in RelaFct"
                   RVar rv ->
                      case evr of
                         ArgR ar ->
                            interpretRelaTerm m
                               (evs,vvs,(rv,interpretRelaTerm m env ar):rvs) rt1
                         _       -> error "wrong argument in RelaFct"

interpretRelaTerms :: Model -> Environment -> [RelaTerm] -> [   [[Bool]]   ]
interpretRelaTerms m env rts =
   map (interpretRelaTerm m env) rts

interpretPartTerm :: Model -> Environment -> PartTerm -> BabyMat
interpretPartTerm m env rt =
   let MO _ _ os _ _ rs _ _ _ = m
   in  case rt of
          Lift (RC rc)        -> lifting m env (RC rc)
          Fetus po1 rt1 po2   -> let interpretedRel = interpretRelaTerm m env rt1
                                     structO1 = getObjectSupportStructure m env po1
                                     structO2 = getObjectSupportStructure m env po2
                                 in  BMGlobal structO1 structO2 interpretedRel
          pt1 :*****: pt2 ->  let BMGlobal lrowR lcolR mR =
                                     convertBMToGlobal $ interpretPartTerm m env pt1
                                  BMGlobal lrowS lcolS mS =
                                     convertBMToGlobal $ interpretPartTerm m env pt2
                              in  BMGlobal lrowR lcolS (transpMat $ lubPart
                                     (codOrdSystOfBabyMat (BMGlobal lrowS lcolS mS))
                                                  (transpMat (mR *** mS)))
          pt1 :|||||: pt2 ->  let BMGlobal lrowR lcolR mR =
                                     convertBMToGlobal $ interpretPartTerm m env pt1
                                  BMGlobal lrowS lcolS mS =
                                     convertBMToGlobal $ interpretPartTerm m env pt2
                              in  BMGlobal lrowR lcolS (transpMat $ lubPart
                                     (codOrdSystOfBabyMat (BMGlobal lrowS lcolS mS))
                                                  (transpMat (mR ||| mS)))
          pt1 :&&&&&: pt2 ->  let BMGlobal lrowR lcolR mR =
```

```
                              convertBMToGlobal $ interpretPartTerm m env pt1
                          BMGlobal lrowS lcolS mS =
                              convertBMToGlobal $ interpretPartTerm m env pt2
                      in  BMGlobal lrowR lcolS (transpMat $ glbPart
                              (codOrdSystOfBabyMat (BMGlobal lrowS lcolS mS))
                                                  (transpMat (mR ||| mS)))
NegaPart pt1    ->  negationBM $ interpretPartTerm m env pt1
IdentP o -> let structO1 = getObjectSupportStructure m env o
                falseBoxN r c = replicate r (replicate c False)
                identRow r z =
                   map (\(_,y) -> falseBoxN r y) (zip [1..z-1] structO1)
                   ++ [ident $ (structO1 !! (z-1))] ++
                   (map (falseBoxN r) (drop z structO1))
                mat = map (\(u,v) -> identRow v u) (zip [1..] structO1)
            in  BMMatOfMat mat
NullP    o o'   -> let structO1 = getObjectSupportStructure m env o
                       structO2 = getObjectSupportStructure m env o'
                       falseBoxN r c = replicate r (replicate c False)
                       line r = map (\cc -> falseBoxN
                                       (structO1 !! (r-1)) cc) structO2
                   in  BMMatOfMat (map line [1..length structO1])
UnivP    o o'   -> let structO1 = getObjectSupportStructure m env o
                       structO2 = getObjectSupportStructure m env o'
                       trueBoxN r c = replicate r (replicate c True)
                       line r = map (\cc -> trueBoxN
                                       (structO1 !! (r-1)) cc) structO2
                   in  BMMatOfMat (map line [1..length structO1])
TranspP  pt1    -> transpositionBM $ interpretPartTerm m env pt1
PPi  o o' -> let structO1 = getObjectSupportStructure m env o
                 structO2 = getObjectSupportStructure m env o'
                 trueBoxPiRow rAt cAt i =
                    replicate (2^cAt) $ replicate (i-1) False ++ [True] ++
                                        (replicate (2^rAt - i) False)
                 trueBoxPi rAt cAt =
                    concatMap (trueBoxPiRow rAt cAt) [1..2^rAt]
                 falseBoxPi rAt cAt =
                    replicate (2^(rAt+cAt)) ([True] ++
                    replicate (2^cAt - 1) False)
                 elementTimesList l e = map (\x -> e + x) l
                 listTimesList l1 l2 =
                    concatMap (elementTimesList l2) l1
                 structProd = listTimesList structO1 structO2
                 rrooww (i,rAt) =
                    let before c i = map (\r -> falseBoxPi r c)
                                         (take (i-1) structO1)
                        hit    c i = trueBoxPi rAt c
                        after  c i =
                           map (\r -> falseBoxPi r c) (drop i structO1)
                        result c i =
                           foldr1 (zipWith (++)) $ (before c i) ++
```

```
                                    [hit c i] ++ (after  c i)
                              in  concatMap (\c -> result c i) structO2
                        in  BMGlobal structProd structO1
                                    (concatMap rrooww (zip [1..] structO1))

          PRho o o' -> let structO1 = getObjectSupportStructure m env o
                           structO2 = getObjectSupportStructure m env o'
                           trueBoxRho  rr cc =
                              concat $ replicate (2^rr) (ident (2^cc))
                           falseBoxRho rr cc = replicate (2^(rr+cc)) ([True] ++
                                                replicate (2^cc - 1) False)
                           elementTimesList l e = map (\x -> e + x) l
                           listTimesList l1 l2 = concatMap (elementTimesList l2) l1
                           structProd = listTimesList structO1 structO2
                           rrooww ze sp = foldr1 (zipWith (++)) $
                             (map  (\c -> falseBoxRho (structO1 !! (ze-1))
                              (structO2 !! (c-1))) [1..sp-1]) ++ [trueBoxRho
                              (structO1 !! (ze-1)) (structO2 !! (sp-1))] ++
                              (map  (\c -> falseBoxRho ze (structO2 !! (c-1)))
                                   [sp+1..length structO2])
                           diag ze = concatMap (rrooww ze) [1 .. length structO2]
                       in  BMGlobal structProd structO2
                              (concatMap diag [1..length structO1])
          PIota o o'  ->
             let BMGlobal rId cId mId = convertBMToGlobal $
                                       interpretPartTerm m env $ IdentP o
                 BMGlobal rNu cNu mNu = convertBMToGlobal $
                                       interpretPartTerm m env $ NullP  o o'
             in  BMGlobal rId (cId ++ cNu) (zipWith (++) mId mNu)
          PKappa o o' ->
             let BMGlobal rNu cNu mNu = convertBMToGlobal $
                                       interpretPartTerm m env $ NullP  o' o
                 BMGlobal rId cId mId = convertBMToGlobal $
                                       interpretPartTerm m env $ IdentP o'
             in  BMGlobal rId (cNu ++ cId) (zipWith (++) mNu mId)
          PCASE rt1 rt2 ->
               let BMGlobal r1 c1 m1 = convertBMToGlobal $
                                       interpretPartTerm m env $ rt1
                   BMGlobal r2 _  m2 = convertBMToGlobal $
                                       interpretPartTerm m env $ rt2
               in  BMGlobal (r1 ++ r2) c1 (m1 ++ m2)
          Lift rt1    -> lifting m env rt1


interpretVectFct ::  Model -> Environment -> VectFct -> [Bool] -> [Bool]
interpretVectFct m env vf bv =
   let VFCT vv vt = vf
       (evs,vvs,rvs) = env
       viWITHbv = (evs,(vv,bv) : vvs,rvs)
   in  interpretVectTerm m viWITHbv vt
```

```
interpretRelaFct ::  Model -> Environment -> RelaFct -> EVRArg -> [[Bool]]
interpretRelaFct m env rf bm =
   let RFCT evr rt = rf
       (evs,vvs,rvs) = env
       viWITHbm =
          case evr of
             EVar ea ->
                case bm of
                   EArg eval -> ((ea,eval) : evs,vvs,rvs)
                   _         -> error "RFCT: variable/argument different type"
             VVar va ->
                case bm of
                   VArg vval -> (evs,(va,vval) : vvs,rvs)
                   _         -> error "RFCT: variable/argument different type"
             RVar ra ->
                case bm of
                   RArg rval -> (evs,vvs,(ra,rval) : rvs)
                   _         -> error "RFCT: variable/argument different type"
   in  interpretRelaTerm m viWITHbm rt


interpretElemSET :: Model -> Environment -> ElemSET  -> [Int]
interpretElemSET m env es =
   case es of
      -- VarES s o ->
      ET    o  -> [1..(arity m env o)]
      ES ev fs -> let MO _ _ os _ _ _ _ _ _ = m
                      (a,b,c)  = env
                      size = arity m env $ domEV ev
                      allElems = [1..size]
                      modify list name value =
                         map (\(n,v) -> case n == name of
                                          True  -> (n,value)
                                          False -> (n,v)    ) list
                  in  filter (\eee -> interpretFormulae m
                                        (modify a ev eee, b, c) fs) allElems
      EX ets _ -> map (interpretElemTerm m env) ets
interpretVectSET :: Model -> Environment -> VectSET  -> [[Bool]]
interpretVectSET m env vs =
   case vs of
      -- VarVS s o ->
      VS vv fs -> let MO _ _ os _ _ _ _ _ _ = m
                      (a,b,c)  = env
                      size = arity m env $ domVV vv
                      allVects = [ r | r <- transpMat (epsi size) ]
                      modify list name value =
                         map (\(n,v) -> case n == name of
                                          True  -> (n,value)
                                          False -> (n,v)    ) list
                  in  filter (\vvv -> interpretFormulae m
```

```
                        (a, modify ((vv,  []) : b) vv vvv, c) fs) allVects
      VX vts _ -> map (interpretVectTerm m env) vts


interpretRelaSET :: Model -> Environment -> RelaSET  -> [ [[Bool]] ]
interpretRelaSET m env rs =
   case rs of
      --VarRS s o o'   ->
      RS rv fs   -> let MO _ _ os _ _ _ _ _ _ = m
                        (a,b,c)  = env
                        size = (arity m env $ domRV rv,
                                arity m env $ domRV rv)
                        allRelas = [ r | r <- allNByM (fst size) (snd size) ]
                        modify list name value =
                           map (\(n,v) -> case n == name of
                                             True  -> (n,value)
                                             False -> (n,v)    ) list
                    in  filter (\rrr -> interpretFormulae m
                                          (a, b, modify c rv rrr) fs) allRelas
      RT fER (ET o) ->
        let elemConstSet = map (\i -> EC $ GenericElemNotation o i)
                               [1..(arity m env o)]
            rts = map (RFctAppl fER) (map ArgE elemConstSet)
        in  interpretRelaTerms m env rts
      RX rts _ _ -> map (interpretRelaTerm m env) rts



interpretElemForm :: Model -> Environment -> ElemForm -> Bool
interpretElemForm m env f =
   let MO _ _ os cs vs rs _ _ _  = m
   in  case f of
          Equation t1 t2 -> interpretElemTerm m env t1 ==
                            interpretElemTerm m env t2
          NegaEqua t1 t2 -> interpretElemTerm m env t1 /=
                            interpretElemTerm m env t2
          QuantElemForm q v fs ->
               let VarE s1 o1 = v
                   carriers = [1 .. arity m env o1]
                   (evs,vvs,rvs) = env
                   lWITHOUTv = filter (\(a,b) -> a /= v) evs
                   vis  = map (\p -> ((p : lWITHOUTv))) [ (v, nn) | nn <- carriers ]
                   andORor = case q of
                                Univ -> and
                                Exis -> or
               in  andORor (map (\vi' -> interpretFormulae
                                    m (vi',vvs,rvs) fs) vis)


interpretVectForm :: Model -> Environment -> VectForm -> Bool
interpretVectForm m env vf =
```

```
   let MO _ _ os cs vs rs _ _ _ = m
   in  case vf of
          (vt1 :<===: vt2) -> let boolVect1 = interpretVectTerm m env vt1
                                  boolVect2 = interpretVectTerm m env vt2
                              in  boolVect1 <= boolVect2
          (_ :>===: _)     -> interpretFormula m env $ expandDefinedVectForm vf
          (_ :====: _)     -> interpretFormula m env $ expandDefinedVectForm vf
          (_ :<=/=: _)     -> interpretFormula m env $ expandDefinedVectForm vf
          (_ :>=/=: _)     -> interpretFormula m env $ expandDefinedVectForm vf
          (_ :==/=: _)     -> interpretFormula m env $ expandDefinedVectForm vf
          VE    vt et      -> let intVT = interpretVectTerm m env vt
                                  intET = interpretElemTerm m env et
                              in  intVT !! (intET - 1)
          VectInSet vt vs  ->
             case vs of
                --VarVS s o ->
                VS vv  fs -> let boolVect  = interpretVectTerm m env vt
                                 (evs,vvs,rvs) = env
                                 env' = (evs,(vv,boolVect):vvs,rvs)
                             in  interpretFormulae m env' fs
                VX vts co -> let boolVect  = interpretVectTerm m env vt
                                 boolVects = interpretVectSET  m env vs
                             in  boolVect `elem` boolVects
          QuantVectForm q vv fs ->
               let VarV _ o1 = vv
                   carrierSize = arity m env o1
                   allVects = transpMat (epsi carrierSize)
                   (evs,vvs,rvs) = env
                   lWITHOUTv = filter (\(a,b) -> a /= vv) vvs
                   vis  = map (\p -> ((p : lWITHOUTv)))
                              [ (vv, nn) | nn <- allVects ]
                   andORor = case q of Univ -> and
                                       Exis -> or
               in  andORor (map (\vi' -> interpretFormulae
                                          m (evs,vi',rvs) fs) vis)


interpretRelaForm :: Model -> Environment -> RelaForm -> Bool
interpretRelaForm m env rf =
   case rf of
      (rt1 :<==: rt2) -> let int1 = interpretRelaTerm m env rt1
                             int2 = interpretRelaTerm m env rt2
                         in  int1 <== int2
      _ :>==: _       -> interpretFormula m env $ expandDefinedRelaForm rf
      _ :===: _       -> interpretFormula m env $ expandDefinedRelaForm rf
      _ :<=/: _       -> interpretFormula m env $ expandDefinedRelaForm rf
      _ :>=/: _       -> interpretFormula m env $ expandDefinedRelaForm rf
      _ :=/=: _       -> interpretFormula m env $ expandDefinedRelaForm rf
      (rt1 :<=/: rt2) -> let int1 = interpretRelaTerm m env rt1
                             int2 = interpretRelaTerm m env rt2
```

```
                        in  not (int1 <== int2)
      (rt1 :>=/: rt2) -> let int1 = interpretRelaTerm m env rt1
                             int2 = interpretRelaTerm m env rt2
                        in  not (int2 <== int1)
      REE rt1 et1 et2 -> let intRT1 = interpretRelaTerm m env rt1
                             intET1 = interpretElemTerm m env et1
                             intET2 = interpretElemTerm m env et2
                        in  matAccess intRT1 intET1 intET2
      RelaInSet rt rs ->
         case rs of
            --VarRS s o o' ->
            RS rv  fs     -> let boolMat = interpretRelaTerm m env rt
                                 (evs,vvs,rvs) = env
                                 env' = (evs,vvs,(rv,boolMat):rvs)
                             in  interpretFormulae m env' fs
            RX rts co co' -> let boolMat = interpretRelaTerm m env rt
                                 boolMats = interpretRelaSET m env rs
                             in  boolMat `elem` boolMats
      -- QuantRelaForm q rv fs ->

interpretPartForm :: Model -> Environment -> PartForm -> Bool
interpretPartForm m env rf =
   case rf of
      (rt1 :<====: rt2) -> let bm1 = interpretPartTerm m env rt1
                               bm2 = interpretPartTerm m env rt2
                               BMMatOfMat piBm1 = convertBMToMatOfMat bm1
                               BMMatOfMat piBm2 = convertBMToMatOfMat bm2
                               containedMatOfMatRow mr1 mr2 =
                                  and $ zipWith (<==) mr1 mr2
                               containedMatOfMat m1 m2 =
                                  and $ zipWith containedMatOfMatRow m1 m2
                           in  containedMatOfMat piBm1 piBm2
      (rt1 :>====: rt2) ->       interpretPartForm m env $ rt2 :<====: rt1
      (rt1 :<=/==: rt2) -> not $ interpretPartForm m env $ rt1 :<====: rt2
      (rt1 :>=/==: rt2) -> not $ interpretPartForm m env $ rt2 :<====: rt1
      StrictPartContained pt1 pt2 ->
                           let bm1 = interpretPartTerm m env pt1
                               bm2 = interpretPartTerm m env pt2
                               strictBm1 = strictBabyMat bm1
                               BMMatOfMat piBm1 = convertBMToMatOfMat bm1
                               BMMatOfMat piBm2 =
                                     convertBMToMatOfMat strictBm1
                               containedMatOfMatRow mr1 mr2 =
                                  and $ zipWith (<==) mr1 mr2
                               containedMatOfMat m1 m2 =
                                  and $ zipWith containedMatOfMatRow m1 m2
                           in  containedMatOfMat piBm1 piBm2


interpretFormula :: Model -> Environment -> Formula -> Bool
```

```
interpretFormula m env f =
   let MO _ _ os cs vs rs _ _ _ = m
   in  case f of
          EF ef -> interpretElemForm m env ef
          VF vf -> interpretVectForm m env vf
          RF rf -> interpretRelaForm m env rf
          PF pf -> interpretPartForm m env pf
          Verum          -> True
          Falsum         -> False
          Negated  f1    -> not (interpretFormula m env f1)
          Implies  f1 f2 -> not (interpretFormula m env f1)  ||
                                (interpretFormula m env f2)
          SemEqu   _   _ -> interpretFormula m env $ expandDefinedFormula f
          Conjunct f1 f2 ->     (interpretFormula m env f1)  &&
                                (interpretFormula m env f2)
          Disjunct f1 f2 ->     (interpretFormula m env f1)  ||
                                (interpretFormula m env f2)
interpretFormulae :: Model -> Environment -> [Formula] -> Bool
interpretFormulae m env fs = and (map (interpretFormula m env) fs)
```

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

```
checkIsModelForTheory mo =
   let MO moS th osM csM vsM rsM fsM vfsM rfsM = mo
       TH thS osT csT vsT rsT fsT vfsT rfsT fs = th
       formulaeSatisfied = interpretFormulae mo ([],[],[]) fs
   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))  &&
       (sort fsT  == (sort $ map (\(InterFct co _) -> co) fsM))  &&
       (sort vfsT == (sort $ map (\(InterVFc co _) -> co) vfsM)) &&
       (sort rfsT == (sort $ map (\(InterRFc co _) -> co) rfsM)) &&
       (aritiesConsistent mo) && (namesDisjointM mo) && (namesDisjointT th) &&
       (length vfsT == length vfsM) && (length rfsT == length rfsM) &&
       formulaeSatisfied
```

# 4 Rules and Transformations

In various ways, terms and formulae will be transformed and translated so as to achieve certain goals. This may be part of an attempt to introduce a proof system. Such goals may, however, also try to express a relation formula in element form; it may be to print a formula in readable TEX- or ASCII-form. The following sections present a common basis for that, namely necessary functions around substitution, matching, and unification.

## 4.1 Renaming of Variables

Sometimes it is desirable to rename variables consistently in order to obtain short versions after originally using automated variables. For this, we provide a renaming mechanism.

```
objectVars   = ["O_1","O_2","O_3","O_4","O_5","O_6","O_7"]
relationVars = ["P","Q","R","S","T","A","B","C",
                "P'","Q'","R'","S'","T'","A'","B'","C'"]
vectorVars   = ["u","v","w","t","u'","v'","w'","t'"]
elementVars  = ["a","b","c","d","a'","b'","c'","d'"]
formulaVars  = ["A","B","C","D","A'","B'","C'","D'"]
relaSetVars  = ["{\\cal S}","{\\cal T}","{\\cal U}","{\\cal V}"]
```

It is alway understood, that *all* the syntctical material is determined first. We indicate how, but cannot foresee the necessary typing, so that no function can fully be given. Then the renamings have to be determined zipping with our standard name proposals.

```
determineRenamingLists sM =
   let (a,_,c,_,e,_,g,_,_,_) = sM
       objectZip   = zip objectVars   a
       elementZip  = zip elementVars  c
       vectorZip   = zip vectorVars   e
       relationZip = zip relationVars g
   in  (map (\(f1,e1) -> (e1,VarO f1))                objectZip,
        map (\(f1,e1) -> (e1,VarE f1 (domEV e1)))     elementZip,
        map (\(f1,e1) -> (e1,VarV f1 (domVV e1)))     vectorZip,
        map (\(f1,e1) -> (e1,VarR f1 (domRV e1)
                                     (codRV e1)))     relationZip,[])
renameSingle c = rename rNList c  where rNList = determineRenamingLists $ syntMat c
```

As a basic technique we introduce the possibility to provide lists of pairs for variables ranging over category objects, elements, vectors, and relations.

```
renameCatObjConst :: ([(CatObjVar,CatObjVar)],[(ElemVari,ElemVari)],
                      [(VectVari, VectVari)], [(RelaVari,RelaVari)],
                      [(FormVari, FormVari)]) -> CatObjCst -> CatObjCst
renameCatObjConst l oc = oc

renameCatObjVar :: ([(CatObjVar,CatObjVar)],[(ElemVari,ElemVari)],
                    [(VectVari,VectVari)],[(RelaVari,RelaVari)],
                    [(FormVari,FormVari)]) -> CatObjVar -> CatObjVar
renameCatObjVar l ov =
   let (a,_,_,_,_) = l
       f = filter (\(x,_) -> x == ov) a
   in  snd $ head f

renameCatObject :: ([(CatObjVar,CatObjVar)],[(ElemVari,ElemVari)],
                    [(VectVari,VectVari)],[(RelaVari,RelaVari)],
                    [(FormVari,FormVari)]) -> CatObject -> CatObject
renameCatObject l co =
   case co of
      OC oc -> OC $ renameCatObjConst l oc
      OV ov -> OV $ renameCatObjVar   l ov
      DirPro o1 o2 -> DirPro (renameCatObject l o1)
                             (renameCatObject l o2)
      DirSum o1 o2 -> DirSum (renameCatObject l o1)
                             (renameCatObject l o2)
      DirPow o1    -> DirPow (renameCatObject l o1)
      UnitOb       -> UnitOb
      QuotMod rt -> QuotMod (renameRelaTerm l rt)
      InjFrom vt -> InjFrom (renameVectTerm l vt)
      --Strict  po -> Strict (renameParObject l po)

renameElemConst :: ([(CatObjVar,CatObjVar)],[(ElemVari,ElemVari)],
                    [(VectVari ,VectVari )],[(RelaVari,RelaVari)],
                    [(FormVari ,FormVari)]) -> ElemConst -> ElemConst
renameElemConst l ec =
   case ec of
      Elem s o -> Elem s (renameCatObject l o)

renameElemVari :: ([(CatObjVar,CatObjVar)],[(ElemVari,ElemVari)],
                   [(VectVari ,VectVari )],[(RelaVari,RelaVari)],
                   [(FormVari ,FormVari )]) -> ElemVari -> ElemVari
renameElemVari l ev =
   let (_,b,_,_,_) = l
       evNEU = snd $ head  $ filter (\(x,_) -> x == ev) b
   in  case evNEU of
          VarE        s   o -> VarE s (renameCatObject l o)
          IndexedVarE s i o -> VarE s (renameCatObject l o)

renameElemTerm :: ([(CatObjVar,CatObjVar)],[(ElemVari,ElemVari)],
                   [(VectVari ,VectVari )],[(RelaVari,RelaVari)],
                   [(FormVari ,FormVari )]) -> ElemTerm -> ElemTerm
```

```
renameElemTerm l et =
   case et of
      EC ec -> EC $ renameElemConst l ec
      EV ev -> EV $ renameElemVari  l ev
      Pair et1 et2 -> Pair (renameElemTerm  l et1)
                           (renameElemTerm  l et2)
      Inj1 et o    -> Inj1 (renameElemTerm  l et)
                           (renameCatObject l o)
      Inj2 o  et   -> Inj2 (renameCatObject l o)
                           (renameElemTerm  l et)
      ThatV vt -> ThatV (renameVectTerm l vt)
      SomeV vt -> SomeV (renameVectTerm l vt)
      ThatR rt -> ThatR (renameRelaTerm l rt)
      SomeR rt -> SomeR (renameRelaTerm l rt)
      VectToElem _ -> renameElemTerm l $ expandDefinedElemTerm et
      FuncAppl fc et -> FuncAppl fc (renameElemTerm l et)
      EFctAppl ef et -> EFctAppl (renameElemFct l ef) (renameElemTerm l et)

renameElemTerms :: ([(CatObjVar,CatObjVar)],[(ElemVari,ElemVari)],
                    [(VectVari ,VectVari )],[(RelaVari,RelaVari)],
                    [(FormVari ,FormVari )]) -> [ElemTerm] -> [ElemTerm]
renameElemTerms l ets = map (renameElemTerm l) ets

renameVectConst :: ([(CatObjVar,CatObjVar)],[(ElemVari,ElemVari)],
                    [(VectVari ,VectVari )],[(RelaVari,RelaVari)],
                    [(FormVari ,FormVari )]) -> VectConst -> VectConst
renameVectConst l vc =
   case vc of
      Vect s o -> Vect s (renameCatObject l o)

renameVectVari :: ([(CatObjVar,CatObjVar)],[(ElemVari,ElemVari)],
                   [(VectVari ,VectVari )],[(RelaVari,RelaVari)],
                   [(FormVari ,FormVari )]) -> VectVari -> VectVari
renameVectVari l vv =
   let (_,_,c,_,_) = l
       vvNEU = snd $ head  $ filter (\(x,_) -> x == vv) c
   in  case vvNEU of
          VarV        s   o -> VarV s (renameCatObject l o)
          IndexedVarV s i o -> VarV s (renameCatObject l o)

renameVectTerm :: ([(CatObjVar,CatObjVar)],[(ElemVari,ElemVari)],
                   [(VectVari ,VectVari )],[(RelaVari,RelaVari)],
                   [(FormVari ,FormVari )]) -> VectTerm -> VectTerm
renameVectTerm l vt =
   case vt of
      VC vc -> VC $ renameVectConst l vc
      VV vv -> VV $ renameVectVari  l vv
      rt1 :****: vt2 -> (renameRelaTerm l rt1) :****:
                        (renameVectTerm l vt2)
      vt1 :||||: vt2 -> (renameVectTerm l vt1) :||||:
```

```
                        (renameVectTerm l vt2)
      vt1 :&&&&: vt2 -> (renameVectTerm l vt1) :&&&&:
                        (renameVectTerm l vt2)
      Syq rt1 vt2    -> Syq   (renameRelaTerm l rt1)
                              (renameVectTerm l vt2)
      NegaV     vt1  -> NegaV (renameVectTerm l vt1)
      NullV     o    -> NullV (renameCatObject l o)
      UnivV     o    -> UnivV (renameCatObject l o)
      SupVect   vs   -> SupVect   $ renameVectSET  l vs
      InfVect   vs   -> InfVect   $ renameVectSET  l vs
      RelaToVect rt  -> RelaToVect $ renameRelaTerm l rt
      PointVect et   -> PointVect $ renameElemTerm l et
      PowElemToVect et -> PowElemToVect $ renameElemTerm l et
      --vFct

renameVectTerms :: ([(CatObjVar,CatObjVar)],[(ElemVari,ElemVari)],
                    [(VectVari ,VectVari )],[(RelaVari,RelaVari)],
                    [(FormVari ,FormVari )]) -> [VectTerm] -> [VectTerm]
renameVectTerms l vts = map (renameVectTerm l) vts

renameRelaConst :: ([(CatObjVar,CatObjVar)],[(ElemVari,ElemVari)],
                    [(VectVari ,VectVari )],[(RelaVari,RelaVari)],
                    [(FormVari ,FormVari )]) -> RelaConst -> RelaConst
renameRelaConst l rc =
   case rc of
      Rela s o o' -> Rela s (renameCatObject l o) (renameCatObject l o')

renameRelaVari :: ([(CatObjVar,CatObjVar)],[(ElemVari,ElemVari)],
                   [(VectVari ,VectVari )],[(RelaVari,RelaVari)],
                   [(FormVari ,FormVari )]) -> RelaVari -> RelaVari
renameRelaVari l rv =
   let (a,_,_,d,_) = l
       rvNEU = snd $ head $ filter (\(x,_) -> x == rv) d
   in  case rvNEU of
          VarR        s   o o' -> VarR s (renameCatObject l o)
                                         (renameCatObject l o')
          IndexedVarR s i o o' -> VarR s (renameCatObject l o)
                                         (renameCatObject l o')

renameEVRVari  :: ([(CatObjVar,CatObjVar)],[(ElemVari,ElemVari)],
                    [(VectVari,VectVari)],[(RelaVari,RelaVari)],
                    [(FormVari,FormVari)]) -> EVRVari -> EVRVari
renameEVRVari l evr =
   case evr of
      EVar ev -> EVar $ renameElemVari l ev
      VVar vv -> VVar $ renameVectVari l vv
      RVar rv -> RVar $ renameRelaVari l rv


renameRelaTerm :: ([(CatObjVar,CatObjVar)],[(ElemVari,ElemVari)],
```

```
                  [(VectVari ,VectVari )],[(RelaVari,RelaVari)],
                  [(FormVari ,FormVari )]) -> RelaTerm -> RelaTerm
renameRelaTerm l rt =
   case rt of
      RC rc          -> RC $ renameRelaConst l rc
      RV rv          -> RV $ renameRelaVari l rv
      rt1 :***: rt2 -> (renameRelaTerm l rt1) :***:
                       (renameRelaTerm l rt2)
      rt1 :|||: rt2 -> (renameRelaTerm l rt1) :|||:
                       (renameRelaTerm l rt2)
      rt1 :&&&: rt2 -> (renameRelaTerm l rt1) :&&&:
                       (renameRelaTerm l rt2)
      NegaR     rt1 -> NegaR (renameRelaTerm l rt1)
      Ident     o    -> Ident (renameCatObject   l o )
      NullR     o o' -> NullR (renameCatObject   l o )
                              (renameCatObject   l o')
      UnivR     o o' -> UnivR (renameCatObject   l o )
                              (renameCatObject   l o')
      Convs     rt1 -> Convs (renameRelaTerm l rt1)
      vt :||--: vt' -> (renameVectTerm l vt ) :||--:
                       (renameVectTerm l vt')
      SupRela   rs  -> SupRela $ renameRelaSET  l rs
      InfRela   rs  -> InfRela $ renameRelaSET  l rs
      Pi        o o' -> Pi    (renameCatObject l o)
                              (renameCatObject l o')
      Rho       o o' -> Rho   (renameCatObject l o)
                              (renameCatObject l o')
      (:*:)     _ _  -> renameRelaTerm l $
                           expandDefinedRelaTerm rt
      (:\/:)    _ _  -> renameRelaTerm l $
                           expandDefinedRelaTerm rt
      SyQ       _ _  -> renameRelaTerm l $
                           expandDefinedRelaTerm rt
      Iota      o o' -> Iota  (renameCatObject l o)
                              (renameCatObject l o')
      Kappa     o o' -> Kappa (renameCatObject l o)
                              (renameCatObject l o')
      CASE rt1 rt2  -> CASE (renameRelaTerm l rt1)
                            (renameRelaTerm l rt2)
      Epsi      o    -> Epsi  (renameCatObject l o)
      PointDiag et  -> PointDiag (renameElemTerm l et)
      InjTerm   vt  -> InjTerm (renameVectTerm l  vt)
      ProdVectToRela _ -> renameRelaTerm l $ expandDefinedRelaTerm rt
      --Wait         rt1 -> Wait (substituteInPartTermAccordingToList l rt1)

renameRelaTerms :: ([(CatObjVar,CatObjVar)],[(ElemVari,ElemVari)],
                    [(VectVari ,VectVari )],[(RelaVari,RelaVari)],
                    [(FormVari ,FormVari )]) -> [RelaTerm] -> [RelaTerm]
renameRelaTerms l rts = map (renameRelaTerm l) rts
```

```
renameElemFct :: ([(CatObjVar,CatObjVar)],[(ElemVari,ElemVari)],
                  [(VectVari ,VectVari )],[(RelaVari,RelaVari)],
                  [(FormVari ,FormVari )]) -> ElemFct -> ElemFct
renameElemFct l ef =
   case ef of
      EFCT ev et -> EFCT (renameElemVari l ev) (renameElemTerm l et)

renameVectFct :: ([(CatObjVar,CatObjVar)],[(ElemVari,ElemVari)],
                  [(VectVari ,VectVari )],[(RelaVari,RelaVari)],
                  [(FormVari ,FormVari )]) -> VectFct -> VectFct
renameVectFct l vf =
   case vf of
      VFCT vv vt -> VFCT (renameVectVari l vv) (renameVectTerm l vt)

renameRelaFct :: ([(CatObjVar,CatObjVar)],[(ElemVari,ElemVari)],
                  [(VectVari ,VectVari )],[(RelaVari,RelaVari)],
                  [(FormVari ,FormVari )]) -> RelaFct -> RelaFct
renameRelaFct l rf =
   case rf of
      RFCT evr rt ->
         case evr of
            EVar ea -> RFCT (EVar $ renameElemVari l ea) (renameRelaTerm l rt)
            VVar va -> RFCT (VVar $ renameVectVari l va) (renameRelaTerm l rt)
            RVar ra -> RFCT (RVar $ renameRelaVari l ra) (renameRelaTerm l rt)

renameElemSET :: ([(CatObjVar,CatObjVar)],[(ElemVari,ElemVari)],
                  [(VectVari ,VectVari )],[(RelaVari,RelaVari)],
                  [(FormVari ,FormVari )]) -> ElemSET -> ElemSET
renameElemSET l es =
   case es of
      VarES s o -> VarES s (renameCatObject l o)
      ET      o -> ET (renameCatObject l o)
      ES ev  fs -> ES (renameElemVari l ev) (renameFormulae l fs)
      EX ets o  -> EX (map (renameElemTerm l) ets)
                      (renameCatObject l o)

renameVectSET :: ([(CatObjVar,CatObjVar)],[(ElemVari,ElemVari)],
                  [(VectVari ,VectVari )],[(RelaVari,RelaVari)],
                  [(FormVari ,FormVari )]) -> VectSET -> VectSET
renameVectSET l vs =
   case vs of
      VarVS s o -> VarVS s (renameCatObject l o)
      VS vv  fs -> VS (renameVectVari l vv) (renameFormulae l fs)
      VX vts o  -> VX (map (renameVectTerm l) vts)
                      (renameCatObject l o)

renameRelaSET :: ([(CatObjVar,CatObjVar)],[(ElemVari,ElemVari)],
                  [(VectVari ,VectVari )],[(RelaVari,RelaVari)],
                  [(FormVari ,FormVari )]) -> RelaSET -> RelaSET
renameRelaSET l rs =
```

```
   case rs of
      VarRS s o o' -> VarRS s (renameCatObject l o)
                              (renameCatObject l o')
      RS rv  fs    -> RS (renameRelaVari l rv) (renameFormulae l fs)
      RX  rts o o' -> RX (map (renameRelaTerm l) rts)
                              (renameCatObject l o)
                              (renameCatObject l o')
renameElemForm :: ([(CatObjVar,CatObjVar)],[(ElemVari,ElemVari)],
                   [(VectVari ,VectVari )],[(RelaVari,RelaVari)],
                   [(FormVari ,FormVari )]) -> ElemForm -> ElemForm
renameElemForm l ef =
   case ef of
      Equation et1 et2 -> Equation (renameElemTerm l et1)
                                   (renameElemTerm l et2)
      NegaEqua et1 et2 -> NegaEqua (renameElemTerm l et1)
                                   (renameElemTerm l et2)
      QuantElemForm q ev fs -> QuantElemForm q (renameElemVari l ev)
                                               (renameFormulae l fs)

renameElemForms :: ([(CatObjVar,CatObjVar)],[(ElemVari,ElemVari)],
                    [(VectVari ,VectVari )],[(RelaVari,RelaVari)],
                    [(FormVari ,FormVari )]) -> [ElemForm] -> [ElemForm]
renameElemForms l efs = map (renameElemForm l) efs

renameVectForm :: ([(CatObjVar,CatObjVar)],[(ElemVari,ElemVari)],
                   [(VectVari ,VectVari )],[(RelaVari,RelaVari)],
                   [(FormVari ,FormVari )]) -> VectForm -> VectForm
renameVectForm l vf =
   case vf of
      vt1 :<===: vt2 -> (renameVectTerm l vt1)  :<===:
                        (renameVectTerm l vt2)
      vt1 :>===: vt2 -> (renameVectTerm l vt1)  :>===:
                        (renameVectTerm l vt2)
      vt1 :====: vt2 -> (renameVectTerm l vt1)  :====:
                        (renameVectTerm l vt2)
      vt1 :<=/=: vt2 -> (renameVectTerm l vt1)  :<=/=:
                        (renameVectTerm l vt2)
      vt1 :>=/=: vt2 -> (renameVectTerm l vt1)  :>=/=:
                        (renameVectTerm l vt2)
      vt1 :==/=: vt2 -> (renameVectTerm l vt1)  :==/=:
                        (renameVectTerm l vt2)
      VE        vt et -> VE (renameVectTerm l vt)
                            (renameElemTerm l et)
      VectInSet vt vs -> VectInSet
                            (renameVectTerm l vt)
                            (renameVectSET  l vs)
      QuantVectForm q vv fs -> QuantVectForm q (renameVectVari l vv)
                                   (map (renameFormula l) fs)

renameVectForms :: ([(CatObjVar,CatObjVar)],[(ElemVari,ElemVari)],
```

```
                    [(VectVari ,VectVari )],[(RelaVari,RelaVari)],
                    [(FormVari ,FormVari )]) -> [VectForm] -> [VectForm]
renameVectForms l vfs = map (renameVectForm l) vfs


renameRelaForm :: ([(CatObjVar,CatObjVar)],[(ElemVari,ElemVari)],
                   [(VectVari ,VectVari )],[(RelaVari,RelaVari)],
                   [(FormVari ,FormVari )]) -> RelaForm -> RelaForm
renameRelaForm l rf =
   case rf of
      rt1 :<==: rt2  -> (renameRelaTerm l rt1) :<==:
                        (renameRelaTerm l rt2)
      rt1 :>==: rt2  -> (renameRelaTerm l rt1) :>==:
                        (renameRelaTerm l rt2)
      rt1 :===: rt2  -> (renameRelaTerm l rt1) :===:
                        (renameRelaTerm l rt2)
      rt1 :<=/: rt2  -> (renameRelaTerm l rt1) :<=/:
                        (renameRelaTerm l rt2)
      rt1 :>=/: rt2  -> (renameRelaTerm l rt1) :>=/:
                        (renameRelaTerm l rt2)
      rt1 :=/=: rt2  -> (renameRelaTerm l rt1) :=/=:
                        (renameRelaTerm l rt2)
      RelaInSet rt rs -> RelaInSet
                             (renameRelaTerm l rt)
                             (renameRelaSET  l rs)
      REE rt et1 et2  -> REE (renameRelaTerm l rt)
                             (renameElemTerm l et1)
                             (renameElemTerm l et2)
      QuantRelaForm q rv fs -> QuantRelaForm q (renameRelaVari l rv)
                                    (map (renameFormula l) fs)

renameRelaForms :: ([(CatObjVar,CatObjVar)],[(ElemVari,ElemVari)],
                    [(VectVari ,VectVari )],[(RelaVari,RelaVari)],
                    [(FormVari ,FormVari )]) -> [RelaForm] -> [RelaForm]
renameRelaForms l rfs = map (renameRelaForm l) rfs

renameFormVari :: ([(CatObjVar,CatObjVar)],[(ElemVari,ElemVari)],
                   [(VectVari ,VectVari )],[(RelaVari,RelaVari)],
                   [(FormVari ,FormVari )]) -> FormVari -> FormVari
renameFormVari l fv =
   let (_,_,_,_,f) = l
       fvNEU = snd $ head  $ filter (\(x,_) -> x == fv) f
   in  case fvNEU of
          VarF        s   -> VarF s
          IndexedVarF s i -> VarF s

renameFormula :: ([(CatObjVar,CatObjVar)],[(ElemVari,ElemVari)],
                  [(VectVari ,VectVari )],[(RelaVari,RelaVari)],
                  [(FormVari ,FormVari )]) -> Formula -> Formula
renameFormula l f =
```

```
   case f of
      FV fv -> let (_,_,_,_,fvs) = l
                   ff = filter (\(a,_) -> a == fv) fvs
               in  case null ff of
                      True  -> f
                      False -> FV $ snd (head ff)
      EF ef -> EF (renameElemForm l ef)
      VF vf -> VF (renameVectForm l vf)
      RF rf -> RF (renameRelaForm l rf)
      Verum          -> Verum
      Falsum         -> Falsum
      Negated  f     -> Negated  (renameFormula l f)
      Implies  f f'  -> Implies  (renameFormula l f)
                                 (renameFormula l f')
      SemEqu   f f'  -> SemEqu   (renameFormula l f)
                                 (renameFormula l f')
      Disjunct f f'  -> Disjunct (renameFormula l f)
                                 (renameFormula l f')
      Conjunct f f'  -> Conjunct (renameFormula l f)
                                 (renameFormula l f')

renameFormulae :: ([(CatObjVar,CatObjVar)],[(ElemVari,ElemVari)],
                   [(VectVari ,VectVari )],[(RelaVari,RelaVari)],
                   [(FormVari ,FormVari )]) -> [Formula] -> [Formula]
renameFormulae l fs = map (renameFormula l) fs
```

## 4.2 Substitutions

In all many cases we substitute variables by terms according to a given list of substitutions.

```
substituteInObjectAccordingToList l o =
   case o of
      OC oc -> o
      OV ov -> let (ol,_,_,_,_) = l
                   f = filter (\(a,_) -> a == ov) ol
               in  case null f of
                      True  -> o
                      False -> snd (head f)
      DirPro o1 o2 -> DirPro (substituteInObjectAccordingToList l o1)
                             (substituteInObjectAccordingToList l o2)
      DirSum o1 o2 -> DirSum (substituteInObjectAccordingToList l o1)
                             (substituteInObjectAccordingToList l o2)
      DirPow o1    -> DirPow (substituteInObjectAccordingToList l o1)
      UnitOb       -> o
      QuotMod rt   -> QuotMod (substituteInRelaTermAccordingToList l rt)
      InjFrom vt   -> InjFrom (substituteInVectTermAccordingToList l vt)
      Strict  po   -> Strict  (substituteInParObjAccordingToList   l po)
substituteInParObjAccordingToList l po =
   case po of
```

```
      ParObj co -> ParObj $ substituteInObjectAccordingToList l co
      ParPro po1 po2 -> ParPro (substituteInParObjAccordingToList l po1)
                               (substituteInParObjAccordingToList l po2)
      ParSum po1 po2 -> ParSum (substituteInParObjAccordingToList l po1)
                               (substituteInParObjAccordingToList l po2)
      ParPow po1     -> ParPow (substituteInParObjAccordingToList l po1)


substituteInElemTermAccordingToList l et =
   case et of
      EC ec -> et
      EV ev -> let (_,el,_,_,_) = l
                   f = filter (\(a,_) -> a == ev) el
               in  case null f of
                      True  -> et
                      False -> snd (head f)
      Pair et1 et2 -> Pair (substituteInElemTermAccordingToList l et1)
                           (substituteInElemTermAccordingToList l et2)
      Inj1 et o    -> Inj1 (substituteInElemTermAccordingToList l et)
                           (substituteInObjectAccordingToList   l o)
      Inj2 o  et   -> Inj2 (substituteInObjectAccordingToList   l o)
                           (substituteInElemTermAccordingToList l et)
      ThatV vt -> ThatV (substituteInVectTermAccordingToList l vt)
      SomeV vt -> SomeV (substituteInVectTermAccordingToList l vt)
      ThatR rt -> ThatR (substituteInRelaTermAccordingToList l rt)
      SomeR rt -> SomeR (substituteInRelaTermAccordingToList l rt)
      FuncAppl fc et -> FuncAppl fc (substituteInElemTermAccordingToList l et)
      VectToElem _ -> substituteInElemTermAccordingToList l $
                         expandDefinedElemTerm et
      --      EFctAppl ef et


substituteInVectTermAccordingToList l vt =
   case vt of
      VC _  -> vt
      VV vv -> let (_,_,vl,_,_) = l
                   f = filter (\(a,_) -> a == vv) vl
               in  case null f of
                      True  -> vt
                      False -> snd (head f)
      rt1 :****: vt2 -> (substituteInRelaTermAccordingToList l rt1) :****:
                        (substituteInVectTermAccordingToList l vt2)
      vt1 :||||: vt2 -> (substituteInVectTermAccordingToList l vt1) :||||:
                        (substituteInVectTermAccordingToList l vt2)
      vt1 :&&&&: vt2 -> (substituteInVectTermAccordingToList l vt1) :&&&&:
                        (substituteInVectTermAccordingToList l vt2)
      Syq rt1 vt2    -> Syq   (substituteInRelaTermAccordingToList l rt1)
                              (substituteInVectTermAccordingToList l vt2)
      NegaV     vt1  -> NegaV (substituteInVectTermAccordingToList l vt1)
      NullV     _    -> vt
      UnivV     _    -> vt
```

```
      SupVect   vs   -> SupVect    $ substituteInVectSETAccordingToList  l vs
      InfVect   vs   -> InfVect    $ substituteInVectSETAccordingToList  l vs
      RelaToVect rt  -> RelaToVect $ substituteInRelaTermAccordingToList l rt
      PointVect et   -> PointVect  $ substituteInElemTermAccordingToList l et
      PowElemToVect et -> PowElemToVect $ substituteInElemTermAccordingToList l et
      --vFct

substituteInRelaTermAccordingToList l rt =
   case rt of
      RC _           -> rt
      RV rv          -> let VarR n dom cod = rv
                            (_,_,_,c,_) = l
                            f = filter (\(x,y) -> x == rv) c
                        in  case null f of
                               True  -> rt
                               False -> snd (head f)
      rt1 :***: rt2 -> (substituteInRelaTermAccordingToList l rt1) :***:
                       (substituteInRelaTermAccordingToList l rt2)
      rt1 :|||: rt2 -> (substituteInRelaTermAccordingToList l rt1) :|||:
                       (substituteInRelaTermAccordingToList l rt2)
      rt1 :&&&: rt2 -> (substituteInRelaTermAccordingToList l rt1) :&&&:
                       (substituteInRelaTermAccordingToList l rt2)
      NegaR     rt1 -> NegaR (substituteInRelaTermAccordingToList l rt1)
      Ident     o   -> Ident (substituteInObjectAccordingToList   l o )
      NullR     o o' -> NullR (substituteInObjectAccordingToList   l o )
                              (substituteInObjectAccordingToList   l o')
      UnivR     o o' -> UnivR (substituteInObjectAccordingToList   l o )
                              (substituteInObjectAccordingToList   l o')
      Convs     rt1 -> Convs (substituteInRelaTermAccordingToList l rt1)
      vt :||--: vt' -> (substituteInVectTermAccordingToList l vt ) :||--:
                       (substituteInVectTermAccordingToList l vt')
      SupRela   rs  -> SupRela $ substituteInRelaSETAccordingToList  l rs
      InfRela   rs  -> InfRela $ substituteInRelaSETAccordingToList  l rs
      Pi        o o' -> Pi    (substituteInObjectAccordingToList l o)
                              (substituteInObjectAccordingToList l o')
      Rho       o o' -> Rho   (substituteInObjectAccordingToList l o)
                              (substituteInObjectAccordingToList l o')
      (:*:)     _ _  -> substituteInRelaTermAccordingToList l $
                           expandDefinedRelaTerm rt
      (:\/:)    _ _  -> substituteInRelaTermAccordingToList l $
                           expandDefinedRelaTerm rt
      SyQ       _ _  -> substituteInRelaTermAccordingToList l $
                           expandDefinedRelaTerm rt
      Iota      o o' -> Iota  (substituteInObjectAccordingToList l o)
                              (substituteInObjectAccordingToList l o')
      Kappa     o o' -> Kappa (substituteInObjectAccordingToList l o)
                              (substituteInObjectAccordingToList l o')
      CASE rt1 rt2  -> CASE  (substituteInRelaTermAccordingToList l rt1)
                              (substituteInRelaTermAccordingToList l rt2)
      Epsi      o    -> Epsi  (substituteInObjectAccordingToList l o)
```

```
      PointDiag et  -> PointDiag (substituteInElemTermAccordingToList l et)
      InjTerm   vt  -> InjTerm   (substituteInVectTermAccordingToList l vt)
      ProdVectToRela _ -> substituteInRelaTermAccordingToList l $
                            expandDefinedRelaTerm rt
      --Wait         rt1 -> Wait (substituteInPartTermAccordingToList l rt1)

substituteInPartTermAccordingToList l rt =
   case rt of
      Lift rt1 -> Lift $ substituteInRelaTermAccordingToList l rt1
      Fetus po1 rt1 po2 -> Fetus (substituteInParObjAccordingToList   l po1)
                                 (substituteInRelaTermAccordingToList l rt1)
                                 (substituteInParObjAccordingToList   l po2)


substituteInVectSETAccordingToList l vs =
   case vs of
      VarVS s o -> VarVS s (substituteInObjectAccordingToList l o)
      VS vv  fs -> VS vv (substituteInFormulaeAccordingToList l fs)
                     --assuming that vv is not affected by the substitution
      VX vts o  -> VX (map (substituteInVectTermAccordingToList l) vts)
                           (substituteInObjectAccordingToList l o)

substituteInRelaSETAccordingToList l rs =
   case rs of
      VarRS s o o' -> VarRS s (substituteInObjectAccordingToList l o)
                              (substituteInObjectAccordingToList l o')
      RS rv  fs    -> RS rv (substituteInFormulaeAccordingToList l fs)
                     --assuming that rv is not affected by the substitution
      RX rts o  o' -> RX (map (substituteInRelaTermAccordingToList l) rts)
                              (substituteInObjectAccordingToList l o)
                              (substituteInObjectAccordingToList l o')


substituteInElemFormAccordingToList l ef =
   let (_,b,_,_,_) = l
       b' = map (\(x,EV y) -> (x,y)) b
   in  case ef of
          Equation et1 et2 -> Equation (substituteInElemTermAccordingToList l et1)
                                       (substituteInElemTermAccordingToList l et2)
          NegaEqua et1 et2 -> NegaEqua (substituteInElemTermAccordingToList l et1)
                                       (substituteInElemTermAccordingToList l et2)
          QuantElemForm q ev fs ->
             QuantElemForm q ev (map (substituteInFormulaAccordingToList l) fs)
  -- Check if only those /= ev should be substituted


substituteInVectFormAccordingToList l vf =
   case vf of
      vt1 :<===: vt2 -> (substituteInVectTermAccordingToList l vt1)  :<===:
                        (substituteInVectTermAccordingToList l vt2)
      vt1 :>===: vt2 -> (substituteInVectTermAccordingToList l vt1)  :>===:
```

```
                        (substituteInVectTermAccordingToList l vt2)
      vt1 :====: vt2 -> (substituteInVectTermAccordingToList l vt1)  :====:
                        (substituteInVectTermAccordingToList l vt2)
      vt1 :<=/=: vt2 -> (substituteInVectTermAccordingToList l vt1)  :<=/=:
                        (substituteInVectTermAccordingToList l vt2)
      vt1 :>=/=: vt2 -> (substituteInVectTermAccordingToList l vt1)  :>=/=:
                        (substituteInVectTermAccordingToList l vt2)
      vt1 :==/=: vt2 -> (substituteInVectTermAccordingToList l vt1)  :==/=:
                        (substituteInVectTermAccordingToList l vt2)
      VE        vt et -> VE (substituteInVectTermAccordingToList l vt)
                            (substituteInElemTermAccordingToList l et)
      VectInSet vt vs -> VectInSet
                            (substituteInVectTermAccordingToList l vt)
                            (substituteInVectSETAccordingToList  l vs)
      QuantVectForm q vv fs -> QuantVectForm q vv
                                    (map (substituteInFormulaAccordingToList l) fs)
  -- Check if only those /= ev should be substituted

substituteInRelaFormAccordingToList l rf =
   case rf of
      rt1 :<==: rt2  -> (substituteInRelaTermAccordingToList l rt1) :<==:
                        (substituteInRelaTermAccordingToList l rt2)
      rt1 :>==: rt2  -> (substituteInRelaTermAccordingToList l rt1) :>==:
                        (substituteInRelaTermAccordingToList l rt2)
      rt1 :===: rt2  -> (substituteInRelaTermAccordingToList l rt1) :===:
                        (substituteInRelaTermAccordingToList l rt2)
      rt1 :<=/: rt2  -> (substituteInRelaTermAccordingToList l rt1) :<=/:
                        (substituteInRelaTermAccordingToList l rt2)
      rt1 :>=/: rt2  -> (substituteInRelaTermAccordingToList l rt1) :>=/:
                        (substituteInRelaTermAccordingToList l rt2)
      rt1 :=/=: rt2  -> (substituteInRelaTermAccordingToList l rt1) :=/=:
                        (substituteInRelaTermAccordingToList l rt2)
      RelaInSet rt rs -> RelaInSet
                            (substituteInRelaTermAccordingToList l rt)
                            (substituteInRelaSETAccordingToList  l rs)
      REE rt et1 et2  -> REE (substituteInRelaTermAccordingToList l rt)
                            (substituteInElemTermAccordingToList l et1)
                            (substituteInElemTermAccordingToList l et2)
      QuantRelaForm q rv fs -> QuantRelaForm q rv
                                    (map (substituteInFormulaAccordingToList l) fs)
  -- Check if only those /= ev should be substituted


substituteInFormulaAccordingToList l f =
   case f of
      FV fv -> let (_,_,_,_,fvs) = l
                   ff = filter (\(a,_) -> a == fv) fvs
               in  case null ff of
                      True  -> f
                      False -> snd (head ff)
```

```
      EF ef -> EF (substituteInElemFormAccordingToList l ef)
      VF vf -> VF (substituteInVectFormAccordingToList l vf)
      RF rf -> RF (substituteInRelaFormAccordingToList l rf)
      Verum         -> Verum
      Falsum        -> Falsum
      Negated  f    -> Negated  (substituteInFormulaAccordingToList l f)
      Implies  f f' -> Implies  (substituteInFormulaAccordingToList l f)
                                (substituteInFormulaAccordingToList l f')
      SemEqu   f f' -> SemEqu   (substituteInFormulaAccordingToList l f)
                                (substituteInFormulaAccordingToList l f')
      Disjunct f f' -> Disjunct (substituteInFormulaAccordingToList l f)
                                (substituteInFormulaAccordingToList l f')
      Conjunct f f' -> Conjunct (substituteInFormulaAccordingToList l f)
                                (substituteInFormulaAccordingToList l f')

substituteInFormulaeAccordingToList l fs =
   map (substituteInFormulaAccordingToList l) fs
```

## 4.3 Matching of Terms

Before we define matching and substitution, we will say a word on typing of rules. It is important that one defines a rule with the most general typing. This should later, however, also be applicable to more restricted special cases. To this end one has to do carefully keep track of specialisations occurring when a pattern is matched to a term.

```
type MatchResult = (Bool,([(CatObjVar,CatObject)],[(ElemVari,ElemTerm)],
                          [(VectVari, VectTerm )],[(RelaVari,RelaTerm)],
                          [(FormVari, Formula  )]))

noMatch                      = (False,([],[],[],[],[]))
yesMatch                     = (True, ([],[],[],[],[]))
yesMatchSubstitute quadrupel = (True,quadrupel)
combineMatches m m' = let (a,(l1,l2,l3,l4,l5)) = m
                          (b,(m1,m2,m3,m4,m5)) = m'
                      in  if a && b then (True,(nub (l1 ++ m1),
                                                nub (l2 ++ m2),
                                                nub (l3 ++ m3),
                                                nub (l4 ++ m4),
                                                nub (l5 ++ m5)))
                                    else noMatch
combineMatchSet ms = foldr1 combineMatches ms
```

As one can see, the result of an attempt to match is a quadrupel describing which are the necessary substitutions for the variables. Only when this collection is finished will unification take place.

```
matchOnPatternObject :: CatObject -> CatObject -> MatchResult
matchOnPatternObject op o =
```

```
   case op of
      OC oc  -> case o of
                  OC oc' -> if oc == oc' then yesMatch else noMatch
                  _      -> noMatch
      OV cov -> yesMatchSubstitute ([(cov, o)],[],[],[],[])
      DirPro o1 o2 -> case o of
                        DirPro o1' o2' -> combineMatches
                                            (matchOnPatternObject o1 o1')
                                            (matchOnPatternObject o2 o2')
                        _              -> noMatch
      DirSum o1 o2 -> case o of
                        DirSum o1' o2' -> combineMatches
                                            (matchOnPatternObject o1 o1')
                                            (matchOnPatternObject o2 o2')
                        _              -> noMatch
      DirPow o1    -> case o of
                        DirPow o1'     -> matchOnPatternObject o1 o1'
                        _              -> noMatch
      UnitOb       -> case o of
                        UnitOb         -> yesMatch
                        _              -> noMatch
      -- QuotMod ?
      -- InjFrom ?
      -- Strict  ?
      _                                 -> noMatch

matchOnPatternParObj pop po =
   case pop of
      ParObj cop       -> case po of
                            ParObj co      -> matchOnPatternObject cop co
                            _              -> noMatch
      ParPro pop1 pop2 -> case po of
                            ParPro po1 po2 -> combineMatches
                                                (matchOnPatternParObj pop1 po1)
                                                (matchOnPatternParObj pop2 po2)
                            _              -> noMatch
      ParSum pop1 pop2 -> case po of
                            ParSum po1 po2 -> combineMatches
                                                (matchOnPatternParObj pop1 po1)
                                                (matchOnPatternParObj pop2 po2)
                            _              -> noMatch
      ParPow pop1      -> case po of
                            ParPow po1     -> matchOnPatternParObj pop1 po1
                            _              -> noMatch


matchOnPatternElemConst :: ElemConst -> ElemConst -> MatchResult
matchOnPatternElemConst (Elem sp op) (Elem s o) =
   if sp == s then matchOnPatternObject op o
              else noMatch
```

```
matchOnPatternElemTerm :: ElemTerm -> ElemTerm -> MatchResult
matchOnPatternElemTerm ep et =
   case ep of
      EC ecp    -> case et of
                     EC ec -> matchOnPatternElemConst ecp ec
                     _     -> noMatch
      EV evp    -> combineMatches
                     (matchOnPatternObject (domEV evp) (domET et))
                     (yesMatchSubstitute ([],[(evp, et)],[],[],[]))
      Pair ep1 ep2 ->
             case et of (Pair et1 et2)   -> combineMatches
                                              (matchOnPatternElemTerm ep1 et1)
                                              (matchOnPatternElemTerm ep2 et2)
                        _                -> noMatch
      Inj1 ep op   ->
             case et of (Inj1 et' ot')   -> combineMatches
                                              (matchOnPatternElemTerm ep  et')
                                              (matchOnPatternObject   op  ot')
                        _                -> noMatch
      Inj2 op ep   ->
             case et of (Inj2 ot' et')   -> combineMatches
                                              (matchOnPatternElemTerm ep  et')
                                              (matchOnPatternObject   op  ot')
                        _                -> noMatch
      ThatV vtp -> case et of
                     ThatV vt -> matchOnPatternVectTerm vtp vt
                     _        -> noMatch
      SomeV vtp -> case et of
                     SomeV vt -> matchOnPatternVectTerm vtp vt
                     _        -> noMatch
      ThatR rtp -> case et of
                     ThatR rt -> matchOnPatternRelaTerm rtp rt
                     _        -> noMatch
      SomeR rtp -> case et of
                     SomeR rt -> matchOnPatternRelaTerm rtp rt
                     _        -> noMatch
      FuncAppl fcp etp ->
         case et of
            FuncAppl fc et' -> if fcp /= fc then noMatch
                                            else matchOnPatternElemTerm etp et'
      VectToElem _ -> matchOnPatternElemTerm ep $ expandDefinedElemTerm et
      _               -> noMatch


matchOnPatternVectConst :: VectConst -> VectConst -> MatchResult
matchOnPatternVectConst (Vect sp op) (Vect s o) =
   if sp == s then matchOnPatternObject op o
              else noMatch
```

```
matchOnPatternVectTerm :: VectTerm -> VectTerm -> MatchResult
matchOnPatternVectTerm vp vt =
   case vp of
      VC vcp -> case vt of
                  VC vc -> matchOnPatternVectConst vcp vc
                  _     -> noMatch
      VV vvp -> combineMatches
                  (matchOnPatternObject (domVV vvp) (domVT vt))
                  (yesMatchSubstitute ([],[],[(vvp,vt)],[],[]))
      rtp  :****: vtp ->
             case vt of rt' :****: vt'   -> combineMatches
                                              (matchOnPatternRelaTerm rtp rt')
                                              (matchOnPatternVectTerm vtp vt')
                        _                -> noMatch
      vt1 :||||: vt2 ->
             case vt of vt1' :||||: vt2' -> combineMatches
                                              (matchOnPatternVectTerm vt1 vt1')
                                              (matchOnPatternVectTerm vt2 vt2')
                        _                -> noMatch
      vt1 :&&&&: vt2 ->
             case vt of vt1' :&&&&: vt2' -> combineMatches
                                              (matchOnPatternVectTerm vt1 vt1')
                                              (matchOnPatternVectTerm vt2 vt2')
                        _                -> noMatch
      Syq rt vt1    ->
             case vt of (Syq rt' vt1')   -> combineMatches
                                             (matchOnPatternRelaTerm rt  rt')
                                             (matchOnPatternVectTerm vt1 vt1')
                        _                -> noMatch
      NegaV     vt1 ->
         case vt of NegaV    vt1'        -> matchOnPatternVectTerm vt1 vt1'
                    _                    -> noMatch
      NullV op -> case vt of NullV o     -> matchOnPatternObject op o
                             _           -> noMatch
      UnivV op -> case vt of UnivV o     -> matchOnPatternObject op o
                             _           -> noMatch
      SupVect  vsp    -> case vt of SupVect vs -> matchOnPatternVectSET vsp vs
                                    _          -> noMatch
      InfVect  vsp    -> case vt of InfVect vs -> matchOnPatternVectSET vsp vs
                                    _          -> noMatch
      --RelaToVect rtp ->
      --PointVect  et  ->
      --PowElemToVect et ->
      _                                  -> noMatch

matchOnPatternVectTerms :: [VectTerm] -> [VectTerm] -> MatchResult
matchOnPatternVectTerms vp vts =
   combineMatchSet $ zipWith matchOnPatternVectTerm vp vts

matchOnPatternVectSET :: VectSET -> VectSET -> MatchResult
```

```
matchOnPatternVectSET vsp vs =
   case vsp of
      --VarVS _ _ _    ->
      VS vvp  fsp -> case vs of
                     VS vv fs -> matchOnPatternFormulae fsp fs
                     _        -> noMatch
      VX vtsp op  -> case vs of
                     VX vts o -> combineMatches
                                    (matchOnPatternVectTerms vtsp vts)
                                    (matchOnPatternObject    op   o)
                     _        -> noMatch


matchOnPatternRelaConst :: RelaConst -> RelaConst -> MatchResult
matchOnPatternRelaConst (Rela sp op1 op2) (Rela s o1 o2) =
   if sp == s then combineMatches (matchOnPatternObject op1 o1)
                                  (matchOnPatternObject op2 o2)
              else noMatch

matchOnPatternRelaTerm :: RelaTerm -> RelaTerm -> MatchResult
matchOnPatternRelaTerm rp rt =
   case rp of
      RC rcp  -> case rt of
                    RC rc -> matchOnPatternRelaConst rcp rc
                    _     -> noMatch
      RV rvp  -> combineMatchSet
                    [matchOnPatternObject (domRV rvp) (domRT rt),
                     matchOnPatternObject (codRV rvp) (codRT rt),
                     yesMatchSubstitute ([],[],[],[(rvp,rt)],[])]
      rt1 :***: rt2 ->
         case rt of rt1' :***: rt2'  -> combineMatches
                                          (matchOnPatternRelaTerm rt1 rt1')
                                          (matchOnPatternRelaTerm rt2 rt2')
                    _                -> noMatch
      rt1 :|||: rt2 ->
         case rt of rt1' :|||: rt2'  -> combineMatches
                                          (matchOnPatternRelaTerm rt1 rt1')
                                          (matchOnPatternRelaTerm rt2 rt2')
                    _                -> noMatch
      rt1 :&&&: rt2 ->
         case rt of rt1' :&&&: rt2'  -> combineMatches
                                          (matchOnPatternRelaTerm rt1 rt1')
                                          (matchOnPatternRelaTerm rt2 rt2')
                    _                -> noMatch
      NegaR    rt1 ->
         case rt of NegaR    rt1'    -> matchOnPatternRelaTerm rt1 rt1'
                    _                -> noMatch
      Ident d    ->
         case rt of Ident d'         -> matchOnPatternObject d d'
                    _                -> noMatch
```

```
NullR d c  ->
   case rt of NullR d' c'      -> combineMatches
                                    (matchOnPatternObject d d')
                                    (matchOnPatternObject c c')
               _               -> noMatch
UnivR d c  ->
   case rt of UnivR d' c'      -> combineMatches
                                    (matchOnPatternObject d d')
                                    (matchOnPatternObject c c')
               _               -> noMatch
Convs  rt1 ->
   case rt of Convs  rt1'      -> matchOnPatternRelaTerm rt1 rt1'
               _               -> noMatch
vt1 :||--: vt2 ->
   case rt of vt1' :||--: vt2' -> combineMatches
                                     (matchOnPatternVectTerm vt1 vt1')
                                     (matchOnPatternVectTerm vt2 vt2')
               _               -> noMatch
SupRela rsp ->
   case rt of SupRela rs       -> matchOnPatternRelaSET rsp rs
               _               -> noMatch
InfRela rsp ->
   case rt of InfRela rs       -> matchOnPatternRelaSET rsp rs
               _               -> noMatch
Pi    o o' ->
   case rt of Pi    d' c' -> combineMatches
                                  (matchOnPatternObject o  d')
                                  (matchOnPatternObject o' c')
               _           -> noMatch
Rho   o o' ->
   case rt of Rho   d' c' -> combineMatches
                                  (matchOnPatternObject o  d')
                                  (matchOnPatternObject o' c')
               _           -> noMatch
rt1 :*: rt2 ->
   case rt of rt1' :*: rt2'   -> combineMatches
                                    (matchOnPatternRelaTerm rt1 rt1')
                                    (matchOnPatternRelaTerm rt2 rt2')
               _              -> noMatch
rt1 :\/: rt2 ->
   case rt of rt1' :\/: rt2'  -> combineMatches
                                    (matchOnPatternRelaTerm rt1 rt1')
                                    (matchOnPatternRelaTerm rt2 rt2')
               _              -> noMatch
SyQ rt1 rt2 ->
   case rt of SyQ rt1' rt2'   -> combineMatches
                                    (matchOnPatternRelaTerm rt1 rt1')
                                    (matchOnPatternRelaTerm rt2 rt2')
               _              -> noMatch
Iota  o o' ->
```

```
         case rt of Iota  d' c' -> combineMatches
                                        (matchOnPatternObject o  d')
                                        (matchOnPatternObject o' c')
                   _           -> noMatch
      Kappa o o' ->
         case rt of Kappa d' c' -> combineMatches
                                        (matchOnPatternObject o  d')
                                        (matchOnPatternObject o' c')
                   _           -> noMatch
      CASE rt1 rt2 ->
         case rt of CASE rt1' rt2' -> combineMatches
                                        (matchOnPatternRelaTerm rt1 rt1')
                                        (matchOnPatternRelaTerm rt2 rt2')
                   _            -> noMatch
      Epsi  o    ->
         case rt of Epsi d'     -> matchOnPatternObject o  d'
                   _           -> noMatch
   --PointDiag
   --ProdVectToRela _ ->
   --InjTerm
   --Wait

matchOnPatternPartTerm pp pt =
   case pp of
      Lift rp         -> case pt of
                            Lift rt -> matchOnPatternRelaTerm rp rt
                            _       -> noMatch
      Fetus po1 rt1 po2 ->
         case pt of
            Fetus po1' rt1' po2' -> combineMatchSet
                                       [matchOnPatternParObj   po1 po1',
                                        matchOnPatternRelaTerm rt1 rt1',
                                        matchOnPatternParObj   po2 po2']
            _                    -> noMatch
      pp1 :*****: pp2 ->
         case pt of pt1 :*****: pt2 -> combineMatches
                                        (matchOnPatternPartTerm pp1 pt1)
                                        (matchOnPatternPartTerm pp2 pt2)
                   _               -> noMatch
      pp1 :|||||: pp2 ->
         case pt of pt1 :|||||: pt2 -> combineMatches
                                        (matchOnPatternPartTerm pp1 pt1)
                                        (matchOnPatternPartTerm pp2 pt2)
                   _               -> noMatch
      pp1 :&&&&&: pp2 ->
         case pt of pt1 :&&&&&: pt2 -> combineMatches
                                        (matchOnPatternPartTerm pp1 pt1)
                                        (matchOnPatternPartTerm pp2 pt2)
                   _               -> noMatch
      NegaPart pp1    ->
```

```
   case pt of NegaPart pt1 -> matchOnPatternPartTerm pp1 pt1
              _            -> noMatch
IdentP   po     ->
   case pt of IdentP o     -> matchOnPatternParObj po o
              _            -> noMatch
NullP    po1 po2      ->
   case pt of NullP o1 o2 -> combineMatches
                                    (matchOnPatternParObj po1 o1)
                                    (matchOnPatternParObj po2 o2)
              _           -> noMatch
UnivP    po1 po2      ->
   case pt of UnivP o1 o2 -> combineMatches
                                    (matchOnPatternParObj po1 o1)
                                    (matchOnPatternParObj po2 o2)
              _           -> noMatch
TranspP pp1    ->
   case pt of TranspP pt1 -> matchOnPatternPartTerm pp1 pt1
              _           -> noMatch
PPi    po1 po2      ->
   case pt of PPi o1 o2 -> combineMatches
                                    (matchOnPatternParObj po1 o1)
                                    (matchOnPatternParObj po2 o2)
              _           -> noMatch
PRho   po1 po2      ->
   case pt of PRho o1 o2 -> combineMatches
                                    (matchOnPatternParObj po1 o1)
                                    (matchOnPatternParObj po2 o2)
              _           -> noMatch
pp1 :#: pp2     ->
   case pt of pt1 :#: pt2 -> combineMatches
                                    (matchOnPatternPartTerm pp1 pt1)
                                    (matchOnPatternPartTerm pp2 pt2)
              _               -> noMatch
pp1 :\\//: pp2    ->
   case pt of pt1 :\\//: pt2 -> combineMatches
                                    (matchOnPatternPartTerm pp1 pt1)
                                    (matchOnPatternPartTerm pp2 pt2)
              _               -> noMatch
PIota  po1 po2      ->
   case pt of PIota o1 o2 -> combineMatches
                                    (matchOnPatternParObj po1 o1)
                                    (matchOnPatternParObj po2 o2)
              _           -> noMatch
PKappa po1 po2      ->
   case pt of PKappa o1 o2 -> combineMatches
                                    (matchOnPatternParObj po1 o1)
                                    (matchOnPatternParObj po2 o2)
              _           -> noMatch
PEpsi   po     ->
   case pt of PEpsi o     -> matchOnPatternParObj po o
```

```
                  _           -> noMatch


matchOnPatternRelaTerms :: [RelaTerm] -> [RelaTerm] -> MatchResult
matchOnPatternRelaTerms rp rts =
   combineMatchSet $ zipWith matchOnPatternRelaTerm rp rts

matchOnPatternRelaSET :: RelaSET -> RelaSET -> MatchResult
matchOnPatternRelaSET rsp rs =
   case rsp of
      --VarRS _ _ _    ->
      RS rvp  fsp    -> case rs of
                           RS rv fs    -> matchOnPatternFormulae fsp fs
                           _           -> noMatch
      RX rtsp op op' -> case rs of
                           RX rts o o' -> combineMatchSet
                                             [matchOnPatternRelaTerms rtsp rts,
                                              matchOnPatternObject op  o,
                                              matchOnPatternObject op' o']
                           _           -> noMatch

matchOnPatternPartTerm :: PartTerm -> PartTerm ->
     (Bool,([(CatObjVar,CatObject)],[(ElemVari,ElemTerm)],
            [(VectVari, VectTerm )],[(RelaVari,RelaTerm)],
                         [(FormVari,Formula)]))

matchOnPatternElemForm :: ElemForm -> ElemForm -> MatchResult
matchOnPatternElemForm p ef =
   case p of
      Equation         et1' et2' ->
         case ef of
            Equation et1 et2 -> combineMatches (matchOnPatternElemTerm et1 et1')
                                               (matchOnPatternElemTerm et2 et2')
            _                -> noMatch
      NegaEqua         et1' et2' ->
         case ef of
            NegaEqua et1 et2 -> combineMatches (matchOnPatternElemTerm et1 et1')
                                               (matchOnPatternElemTerm et2 et2')
            _                -> noMatch
      QuantElemForm q evp fsp ->
         case ef of
            QuantElemForm q ev fs -> matchOnPatternFormulae fsp fs
                     --assuming that ev is not affected
            _                          -> noMatch


matchOnPatternVectForm :: VectForm -> VectForm -> MatchResult
matchOnPatternVectForm p vf =
   case p of
      vt1 :<===: vt2 ->
         case vf of
```

```
            vt1' :<===: vt2' -> combineMatches (matchOnPatternVectTerm vt1 vt1')
                                               (matchOnPatternVectTerm vt2 vt2')
            _                -> noMatch
      vt1 :>===: vt2 ->
         case vf of
            vt1' :>===: vt2' -> combineMatches (matchOnPatternVectTerm vt1 vt1')
                                               (matchOnPatternVectTerm vt2 vt2')
            _                -> noMatch
      vt1 :====: vt2 ->
         case vf of
            vt1' :====: vt2' -> combineMatches (matchOnPatternVectTerm vt1 vt1')
                                               (matchOnPatternVectTerm vt2 vt2')
            _                -> noMatch

      vt1 :<=/=: vt2 ->
         case vf of
            vt1' :<=/=: vt2' -> combineMatches (matchOnPatternVectTerm vt1 vt1')
                                               (matchOnPatternVectTerm vt2 vt2')
            _                -> noMatch
      vt1 :>=/=: vt2 ->
         case vf of
            vt1' :>=/=: vt2' -> combineMatches (matchOnPatternVectTerm vt1 vt1')
                                               (matchOnPatternVectTerm vt2 vt2')
            _                -> noMatch
      VE vtp etp ->
         case vf of
            VE vt et         -> combineMatches (matchOnPatternVectTerm vtp vt)
                                               (matchOnPatternElemTerm etp et)
            _                -> noMatch
      VectInSet vt vs ->
         case vf of
            VectInSet vt' vs' -> combineMatches (matchOnPatternVectTerm vt vt')
                                                (matchOnPatternVectSET  vs vs')
            _                  -> noMatch
      --UnivQuantVectForm  vv vf ->
      --ExistQuantVectForm vv vf ->


matchOnPatternRelaForm :: RelaForm -> RelaForm -> MatchResult
matchOnPatternRelaForm p rf =
   case p of
      rt1 :<==: rt2 ->
         case rf of
            rt1' :<==: rt2' -> combineMatches (matchOnPatternRelaTerm rt1 rt1')
                                              (matchOnPatternRelaTerm rt2 rt2')
            _               -> noMatch
      rt1 :>==: rt2 ->
         case rf of
            rt1' :>==: rt2' -> combineMatches (matchOnPatternRelaTerm rt1 rt1')
                                              (matchOnPatternRelaTerm rt2 rt2')
```

```
            _                -> noMatch
      rt1 :===: rt2 ->
         case rf of
            rt1' :===: rt2' -> combineMatches (matchOnPatternRelaTerm rt1 rt1')
                                              (matchOnPatternRelaTerm rt2 rt2')
            _                -> noMatch
      rt1 :<=/: rt2 ->
         case rf of
            rt1' :<=/: rt2' -> combineMatches (matchOnPatternRelaTerm rt1 rt1')
                                              (matchOnPatternRelaTerm rt2 rt2')
            _                -> noMatch
      rt1 :>=/: rt2 ->
         case rf of
            rt1' :>=/: rt2' -> combineMatches (matchOnPatternRelaTerm rt1 rt1')
                                              (matchOnPatternRelaTerm rt2 rt2')
            _                -> noMatch
      RelaInSet rt rs ->
         case rf of
            RelaInSet rt' rs'  -> combineMatches (matchOnPatternRelaTerm rt rt')
                                                 (matchOnPatternRelaSET  rs rs')
            _                  -> noMatch
      REE rt1 et1 et2 ->
         case rf of
            REE rt1' et1' et2' -> combineMatchSet [matchOnPatternRelaTerm rt1 rt1',
                                                   matchOnPatternElemTerm et1 et1',
                                                   matchOnPatternElemTerm et2 et2']
            _                  -> noMatch
      --UnivQuantRelaForm  rv rf ->
      --ExistQuantRelaForm rv rf ->

matchOnPatternPartForm pp pf =
   case pp of
      pt1 :<====: pt2 ->
         case pf of
            pt1' :<====: pt2' -> combineMatches (matchOnPatternPartTerm pt1 pt1')
                                                (matchOnPatternPartTerm pt2 pt2')
            _                 -> noMatch
      pt1 :>====: pt2 ->
         case pf of
            pt1' :>====: pt2' -> combineMatches (matchOnPatternPartTerm pt1 pt1')
                                                (matchOnPatternPartTerm pt2 pt2')
            _                 -> noMatch
      pt1 :<=/==: pt2 ->
         case pf of
            pt1' :<=/==: pt2' -> combineMatches (matchOnPatternPartTerm pt1 pt1')
                                                (matchOnPatternPartTerm pt2 pt2')
            _                 -> noMatch
      pt1 :>=/==: pt2 ->
         case pf of
            pt1' :>=/==: pt2' -> combineMatches (matchOnPatternPartTerm pt1 pt1')
```

```
                                                (matchOnPatternPartTerm pt2 pt2')
            _                   -> noMatch


matchOnPatternFormula :: Formula -> Formula -> MatchResult
matchOnPatternFormula p f =
   case p of
      FV fv -> (yesMatchSubstitute ([],[],[],[],[(fv,f)]))
      EF ef -> case f of EF ef1 -> matchOnPatternElemForm ef ef1
                         _      -> noMatch
      VF vf -> case f of VF vf1 -> matchOnPatternVectForm vf vf1
                         _      -> noMatch
      RF rf -> case f of RF rf1 -> matchOnPatternRelaForm rf rf1
                         _      -> noMatch
      --PF pf -> case f of PF pf1 -> matchOnPatternPartForm pf pf1
      --                   _      -> noMatch
      Verum                     ->
             case f of Verum            -> yesMatch
                       _                -> noMatch
      Falsum    ->
             case f of Falsum           -> yesMatch
                       _                -> noMatch
      Negated  f1    ->
             case f of Negated  f1'     -> matchOnPatternFormula f1 f1'
                       _                -> noMatch
      Implies  f1 f2 ->
             case f of Implies f1' f2'  -> combineMatches
                                                (matchOnPatternFormula f1 f1')
                                                (matchOnPatternFormula f2 f2')
                       _                -> noMatch
      SemEqu   f1 f2 ->
             case f of SemEqu  f1' f2'  -> combineMatches
                                                (matchOnPatternFormula f1 f1')
                                                (matchOnPatternFormula f2 f2')
                       _                -> noMatch
      Disjunct f1 f2 ->
             case f of Disjunct f1' f2' -> combineMatches
                                                (matchOnPatternFormula f1 f1')
                                                (matchOnPatternFormula f2 f2')
                       _                -> noMatch
      Conjunct f1 f2 ->
             case f of Conjunct f1' f2' -> combineMatches
                                                (matchOnPatternFormula f1 f1')
                                                (matchOnPatternFormula f2 f2')
                       _                -> noMatch

matchOnPatternFormulae :: [Formula] -> [Formula] -> MatchResult
matchOnPatternFormulae p fs =
   combineMatchSet $ zipWith matchOnPatternFormula p fs
```

## 4.4 Unification of Types

Unification is related to pattern matching, but tries to substitute on both sides. Nevertheless, unification is not just a symmetric form of matching. When matching patterns one will first collect a set of type restrictions, then try to satisfy them jointly by unification, and finally match in the non-symmetric way as before.

### 4.4.1 Collecting Type Restrictions

When comparing category objects, one may find out that they cannot be made equal, in which case `Nothing` is returned. If they are identical, `Just []` is returned. In other cases, `Just` is returned with a list of category object pairs that need to be unified.

```
compareObjects :: CatObject -> CatObject -> Maybe [(CatObject,CatObject)]
compareObjects o o' =
   case o of
      OC coc       -> case o' of
                        OC coc' -> if coc == coc' then Just []
                                                  else Nothing
                        OV _    -> Just [(o', o)]
                        _       -> Nothing
      OV cov       ->
         case o' of
            OV cov' -> if cov == cov' then Just []
                                      else Just [(o, o')]
            _       -> Just [(o, o')]


      DirPro o1 o2 ->
         case o' of
            DirPro o1' o2' -> collectTypeRestrictionMaybes
                                 [compareObjects o1 o1',compareObjects o2 o2']
            OV _           -> Just [(o', o)]
            _              -> Nothing
      DirSum o1 o2 ->
         case o' of
            DirSum o1' o2' -> collectTypeRestrictionMaybes
                                 [compareObjects o1 o1',compareObjects o2 o2']
            OV _           -> Just [(o', o)]
            _              -> Nothing
      DirPow o1    ->
         case o' of
            DirPow o1'     -> compareObjects o1 o1'
            OV _           -> Just [(o', o)]
            _              -> Nothing
      UnitOb       ->
         case o' of
            UnitOb         -> Just []
            OV _           -> Just [(o', o)]
            _              -> Nothing
```

```
      -- QuotMod
      -- InjFrom
      -- Strict

comparePartialities :: ParObject -> ParObject -> Maybe [(CatObject,CatObject)]
comparePartialities po po' =
   case po of
      ParObj co    -> case po' of
                        ParObj co' -> compareObjects co co'
                        _          -> Nothing
      ParPro o1 o2 ->
         case po' of
            ParPro o1' o2' -> collectTypeRestrictionMaybes
                                 [comparePartialities o1 o1',
                                  comparePartialities o2 o2']
            _              -> Nothing
      ParSum o1 o2 ->
         case po' of
            ParSum o1' o2' -> collectTypeRestrictionMaybes
                                 [comparePartialities o1 o1',
                                  comparePartialities o2 o2']
            _              -> Nothing
      ParPow o1    ->
         case po' of
            ParPow o1'     -> comparePartialities o1 o1'
            _              -> Nothing
```

The following function serves to collect all the requirements for unification that arise from a term as it is built.

```
collectTypeRestrictionMaybes ::
   [Maybe [(CatObject,CatObject)]] -> Maybe [(CatObject,CatObject)]
collectTypeRestrictionMaybes [] = Just []
collectTypeRestrictionMaybes (h:t) =
   case h of
      Just x  -> let trRest = collectTypeRestrictionMaybes t
                 in  case trRest of
                        Just y  -> Just (nub $ x ++ y)
                        Nothing -> Nothing
      Nothing -> Nothing



typeRestrInducedByElemTerm :: ElemTerm -> Maybe [(CatObject,CatObject)]
typeRestrInducedByElemTerm et =
   case et of
      EC (Elem _ _) -> Just []
      EV (VarE _ _) -> Just []
      EV (IndexedVarE _ _ _) -> Just []
```

```
      Pair et1 et2   -> collectTypeRestrictionMaybes
                            [typeRestrInducedByElemTerm et1,
                             typeRestrInducedByElemTerm et2]
      Inj1 et  _     -> typeRestrInducedByElemTerm et
      Inj2 _   et    -> typeRestrInducedByElemTerm et
      ThatV vt       -> typeRestrInducedByVectTerm vt
      SomeV vt       -> typeRestrInducedByVectTerm vt
      ThatR rt       -> typeRestrInducedByRelaTerm rt
      SomeR rt       -> typeRestrInducedByRelaTerm rt
      FuncAppl fc et' -> collectTypeRestrictionMaybes
                            [compareObjects (domFC fc) (domET et'),
                             typeRestrInducedByElemTerm et]
      VectToElem _ -> typeRestrInducedByElemTerm $ expandDefinedElemTerm et

typeRestrInducedByVectTerm :: VectTerm -> Maybe [(CatObject,CatObject)]
typeRestrInducedByVectTerm vt =
   case vt of
      VC _             -> Just []
      VV _             -> Just []
      rt1 :****: vt2  -> collectTypeRestrictionMaybes
                            [compareObjects (codRT rt1) (domVT vt2),
                             typeRestrInducedByRelaTerm rt1,
                             typeRestrInducedByVectTerm vt2]
      vt1 :||||: vt2  -> collectTypeRestrictionMaybes
                            [compareObjects (domVT vt1) (domVT vt2),
                             typeRestrInducedByVectTerm vt1,
                             typeRestrInducedByVectTerm vt2]
      vt1 :&&&&: vt2  -> collectTypeRestrictionMaybes
                            [compareObjects (domVT vt1) (domVT vt2),
                             typeRestrInducedByVectTerm vt1,
                             typeRestrInducedByVectTerm vt2]
      Syq rt1    vt2  -> collectTypeRestrictionMaybes
                            [compareObjects (domRT rt1) (domVT vt2),
                             typeRestrInducedByRelaTerm rt1,
                             typeRestrInducedByVectTerm vt2]
      NegaV     vt1   -> typeRestrInducedByVectTerm vt1
      NullV     _     -> Just []
      UnivV     _     -> Just []
      SupVect   vs    -> typeRestrInducedByVectSet  vs
      InfVect   vs    -> typeRestrInducedByVectSet  vs
      PointVect et    -> typeRestrInducedByElemTerm et
      PowElemToVect et -> typeRestrInducedByElemTerm et
      RelaToVect rt   -> typeRestrInducedByRelaTerm rt
      VFctAppl vf vt2 -> collectTypeRestrictionMaybes
                            [compareObjects (fst $ typeOfFV vf) (domVT vt2),
                             typeRestrInducedByVectFct  vf,
                             typeRestrInducedByVectTerm vt2]


typeRestrInducedByVectTerms :: [VectTerm] -> Maybe [(CatObject,CatObject)]
```

```
typeRestrInducedByVectTerms vts =
   collectTypeRestrictionMaybes (map typeRestrInducedByVectTerm vts)

typeRestrInducedByRelaTerm :: RelaTerm -> Maybe [(CatObject,CatObject)]
typeRestrInducedByRelaTerm rt =
   case rt of
      RC _           -> Just []
      RV _           -> Just []
      rt1 :***:  rt2 -> collectTypeRestrictionMaybes
                           [compareObjects (codRT rt1) (domRT rt2),
                            typeRestrInducedByRelaTerm rt1,
                            typeRestrInducedByRelaTerm rt2]
      rt1 :|||:  rt2 -> collectTypeRestrictionMaybes
                           [compareObjects (domRT rt1) (domRT rt2),
                            compareObjects (codRT rt1) (codRT rt2),
                            typeRestrInducedByRelaTerm rt1,
                            typeRestrInducedByRelaTerm rt2]
      rt1 :&&&:  rt2 -> collectTypeRestrictionMaybes
                           [compareObjects (domRT rt1) (domRT rt2),
                            compareObjects (codRT rt1) (codRT rt2),
                            typeRestrInducedByRelaTerm rt1,
                            typeRestrInducedByRelaTerm rt2]
      NegaR      rt1 -> typeRestrInducedByRelaTerm rt1
      Ident      o   -> collectTypeRestrictionMaybes
                           [compareObjects (domRT rt) (codRT rt),
                            compareObjects (domRT rt) o]
      NullR      _ _ -> Just []
      UnivR      _ _ -> Just []
      Convs      rt1 -> typeRestrInducedByRelaTerm rt1
      vt :||--:  vt' -> collectTypeRestrictionMaybes
                           [typeRestrInducedByVectTerm vt,
                            typeRestrInducedByVectTerm vt']
      SupRela    rs  -> typeRestrInducedByRelaSet  rs
      InfRela    rs  -> typeRestrInducedByRelaSet  rs
      Pi       _ _   -> Just []
      Rho      _ _   -> Just []
      (:*:)  rt1 rt2 -> typeRestrInducedByRelaTerm $ expandDefinedRelaTerm rt
      (:\/:) rt1 rt2 -> typeRestrInducedByRelaTerm $ expandDefinedRelaTerm rt
      SyQ    rt1 rt2 -> typeRestrInducedByRelaTerm $ expandDefinedRelaTerm rt
      Iota     _ _   -> Just []
      Kappa    _ _   -> Just []
      CASE  rt1 rt2  -> collectTypeRestrictionMaybes
                           [compareObjects (codRT rt1) (codRT rt2),
                            typeRestrInducedByRelaTerm rt1,
                            typeRestrInducedByRelaTerm rt2]
      Epsi     _     -> Just []
      PointDiag  et  -> typeRestrInducedByElemTerm et
      ProdVectToRela _ -> typeRestrInducedByRelaTerm $ expandDefinedRelaTerm rt
      InjTerm    vt  -> typeRestrInducedByVectTerm vt
      --Wait       rt1 -> typeRestrInducedByPartTerm rt1
```

```
      Belly pt         -> typeRestrInducedByPartTerm pt
      PartOrd _       -> Just []
      RFctAppl rf rt2 ->
         case rt2 of
            ArgE ae -> collectTypeRestrictionMaybes
                           [compareObjects (fst $ fst $ typeOfFR rf) (domET ae),
                            typeRestrInducedByRelaFct  rf,
                            typeRestrInducedByElemTerm ae]
            ArgV av -> collectTypeRestrictionMaybes
                           [compareObjects (fst $ fst $ typeOfFR rf) (domVT av),
                            typeRestrInducedByRelaFct  rf,
                            typeRestrInducedByVectTerm av]
            ArgR ar -> collectTypeRestrictionMaybes
                           [compareObjects (fst $ fst $ typeOfFR rf) (domRT ar),
                            compareObjects (snd $ fst $ typeOfFR rf) (codRT ar),
                            typeRestrInducedByRelaFct  rf,
                            typeRestrInducedByRelaTerm ar]

typeRestrInducedByRelaTerms :: [RelaTerm] -> Maybe [(CatObject,CatObject)]
typeRestrInducedByRelaTerms rts =
   collectTypeRestrictionMaybes (map typeRestrInducedByRelaTerm rts)

typeRestrInducedByPartTerm :: PartTerm -> Maybe [(CatObject,CatObject)]
typeRestrInducedByPartTerm pt =
   case pt of
      Lift   pt1      -> typeRestrInducedByRelaTerm pt1
      Fetus po1 rt1 po2 ->collectTypeRestrictionMaybes
                            [compareObjects (domRT $ PartOrd po1) (domRT rt1),
                             typeRestrInducedByRelaTerm rt1,
                             compareObjects (domRT $ PartOrd po2) (codRT rt1)]
      pt1 :*****: pt2 -> collectTypeRestrictionMaybes
                           [comparePartialities (codPT pt1) (domPT pt2),
                            typeRestrInducedByPartTerm pt1,
                            typeRestrInducedByPartTerm pt2]
      pt1 :|||||: pt2 -> collectTypeRestrictionMaybes
                           [comparePartialities (domPT pt1) (domPT pt2),
                            comparePartialities (codPT pt1) (codPT pt2),
                            typeRestrInducedByPartTerm pt1,
                            typeRestrInducedByPartTerm pt2]
      pt1 :&&&&&: pt2 -> collectTypeRestrictionMaybes
                           [comparePartialities (domPT pt1) (domPT pt2),
                            comparePartialities (codPT pt1) (codPT pt2),
                            typeRestrInducedByPartTerm pt1,
                            typeRestrInducedByPartTerm pt2]
      NegaPart pt1    -> typeRestrInducedByPartTerm pt1
      IdentP   po     -> Just []
      NullP    d   c  -> Just []
      UnivP    d   c  -> Just []
      TranspP  pt1    -> typeRestrInducedByPartTerm pt1
      PPi      _   _  -> Just []
```

```
      PRho     _   _  -> Just []
      pt1 :#: pt2     -> collectTypeRestrictionMaybes
                           [typeRestrInducedByPartTerm pt1,
                            typeRestrInducedByPartTerm pt2]
      pt1 :\\//: pt2  -> collectTypeRestrictionMaybes
                           [comparePartialities (domPT pt1) (domPT pt2),
                            typeRestrInducedByPartTerm pt1,
                            typeRestrInducedByPartTerm pt2]
      PIota    _ _    -> Just []
      PKappa   _ _    -> Just []
      PEpsi    _      -> Just []


typeRestrInducedByVectSet :: VectSET -> Maybe [(CatObject,CatObject)]
typeRestrInducedByVectSet vs =
   case vs of
      VarVS _ _ -> Just []
      VS _   fs -> typeRestrInducedByFormulae  fs
      VX vts o  -> case vts == [] of
                     True  -> Just []
                     False -> collectTypeRestrictionMaybes
                                [compareObjects (domVT $ head vts) o,
                                 typeRestrInducedByVectTerms vts]


typeRestrInducedByVectFct :: VectFct -> Maybe [(CatObject,CatObject)]
typeRestrInducedByVectFct (VFCT vv vt) = typeRestrInducedByVectTerm vt

typeRestrInducedByRelaFct :: RelaFct -> Maybe [(CatObject,CatObject)]
typeRestrInducedByRelaFct (RFCT rv rt) = typeRestrInducedByRelaTerm rt

typeRestrInducedByRelaSet :: RelaSET -> Maybe [(CatObject,CatObject)]
typeRestrInducedByRelaSet rs =
   case rs of
      VarRS _ o o' -> Just []
      RS _   fs    -> typeRestrInducedByFormulae  fs
      RX rts o o'  ->
       case rts == [] of
          True  -> Just []
          False -> collectTypeRestrictionMaybes
                    ((map (\x -> compareObjects o  (domRT x)) rts)  ++
                     (map (\x -> compareObjects o' (codRT x)) rts)  ++
                     [typeRestrInducedByRelaTerms rts])
                    --[compareObjects (domRT $ head rts) o,
                    -- compareObjects (codRT $ head rts) o',
                    -- typeRestrInducedByRelaTerms rts]

typeRestrInducedByElemForm :: ElemForm -> Maybe [(CatObject,CatObject)]
typeRestrInducedByElemForm ef =
   case ef of
      Equation           et et' -> collectTypeRestrictionMaybes
```

```
                                  [typeRestrInducedByElemTerm et,
                                   typeRestrInducedByElemTerm et']
      NegaEqua           et et' -> collectTypeRestrictionMaybes
                                  [typeRestrInducedByElemTerm et,
                                   typeRestrInducedByElemTerm et']
      QuantElemForm _ ev efs ->  typeRestrInducedByFormulae efs


typeRestrInducedByVectForm :: VectForm -> Maybe [(CatObject,CatObject)]
typeRestrInducedByVectForm vf =
   case vf of
      vt1 :<===: vt2  -> collectTypeRestrictionMaybes
                            [compareObjects (domVT vt1) (domVT vt2),
                             typeRestrInducedByVectTerm vt1,
                             typeRestrInducedByVectTerm vt2]
      vt1 :>===: vt2  -> collectTypeRestrictionMaybes
                            [compareObjects (domVT vt1) (domVT vt2),
                             typeRestrInducedByVectTerm vt1,
                             typeRestrInducedByVectTerm vt2]
      vt1 :====: vt2  -> collectTypeRestrictionMaybes
                            [compareObjects (domVT vt1) (domVT vt2),
                             typeRestrInducedByVectTerm vt1,
                             typeRestrInducedByVectTerm vt2]
      vt1 :<=/=: vt2  -> collectTypeRestrictionMaybes
                            [compareObjects (domVT vt1) (domVT vt2),
                             typeRestrInducedByVectTerm vt1,
                             typeRestrInducedByVectTerm vt2]
      vt1 :>=/=: vt2  -> collectTypeRestrictionMaybes
                            [compareObjects (domVT vt1) (domVT vt2),
                             typeRestrInducedByVectTerm vt1,
                             typeRestrInducedByVectTerm vt2]
      vt1 :==/=: vt2  -> collectTypeRestrictionMaybes
                            [compareObjects (domVT vt1) (domVT vt2),
                             typeRestrInducedByVectTerm vt1,
                             typeRestrInducedByVectTerm vt2]
      VE  vt et       -> collectTypeRestrictionMaybes
                            [compareObjects (domVT vt) (domET et),
                             typeRestrInducedByVectTerm vt,
                             typeRestrInducedByElemTerm et]
      VectInSet vt vs -> collectTypeRestrictionMaybes
                            [compareObjects (domVT vt) (domVS vs),
                             typeRestrInducedByVectTerm vt,
                             typeRestrInducedByVectSet  vs]
      QuantVectForm _ vv fs -> typeRestrInducedByFormulae fs


typeRestrInducedByRelaForm :: RelaForm -> Maybe [(CatObject,CatObject)]
typeRestrInducedByRelaForm rf =
   case rf of
      rt1 :<==: rt2   -> collectTypeRestrictionMaybes
                           [compareObjects (domRT rt1) (domRT rt2),
```

```
                          compareObjects (codRT rt1) (codRT rt2),
                          typeRestrInducedByRelaTerm rt1,
                          typeRestrInducedByRelaTerm rt2]
      rt1 :>==: rt2   -> typeRestrInducedByFormula $ expandDefinedRelaForm rf
      rt1 :===: rt2   -> typeRestrInducedByFormula $ expandDefinedRelaForm rf
      rt1 :<=/: rt2   -> collectTypeRestrictionMaybes
                         [compareObjects (domRT rt1) (domRT rt2),
                          compareObjects (codRT rt1) (codRT rt2),
                          typeRestrInducedByRelaTerm rt1,
                          typeRestrInducedByRelaTerm rt2]
      rt1 :>=/: rt2   -> collectTypeRestrictionMaybes
                         [compareObjects (domRT rt1) (domRT rt2),
                          compareObjects (codRT rt1) (codRT rt2),
                          typeRestrInducedByRelaTerm rt1,
                          typeRestrInducedByRelaTerm rt2]
      rt1 :=/=: rt2   -> collectTypeRestrictionMaybes
                         [compareObjects (domRT rt1) (domRT rt2),
                          compareObjects (codRT rt1) (codRT rt2),
                          typeRestrInducedByRelaTerm rt1,
                          typeRestrInducedByRelaTerm rt2]
      REE rt et1 et2  -> collectTypeRestrictionMaybes
                         [compareObjects (domRT rt) (domET et1),
                          compareObjects (codRT rt) (domET et2),
                          typeRestrInducedByRelaTerm rt,
                          typeRestrInducedByElemTerm et1,
                          typeRestrInducedByElemTerm et2]
      RelaInSet rt rs -> collectTypeRestrictionMaybes
                         [compareObjects (domRT rt) (domRS rs),
                          compareObjects (codRT rt) (codRS rs),
                          typeRestrInducedByRelaTerm rt,
                          typeRestrInducedByRelaSet  rs]
      QuantRelaForm _ rv fs -> typeRestrInducedByFormulae fs

typeRestrInducedByRelaForms :: [RelaForm] -> Maybe [(CatObject,CatObject)]
typeRestrInducedByRelaForms rfs =
   collectTypeRestrictionMaybes (map typeRestrInducedByRelaForm rfs)

typeRestrInducedByPartForm :: PartForm -> Maybe [(CatObject,CatObject)]
typeRestrInducedByPartForm pf =
   case pf of
      pt1 :<====: pt2   -> collectTypeRestrictionMaybes
                           [comparePartialities (domPT pt1) (domPT pt2),
                            comparePartialities (codPT pt1) (codPT pt2),
                            typeRestrInducedByPartTerm pt1,
                            typeRestrInducedByPartTerm pt2]
      pt1 :>====: pt2   -> collectTypeRestrictionMaybes
                           [comparePartialities (domPT pt1) (domPT pt2),
                            comparePartialities (codPT pt1) (codPT pt2),
                            typeRestrInducedByPartTerm pt1,
                            typeRestrInducedByPartTerm pt2]
```

```
      pt1 :<=/==: pt2   -> collectTypeRestrictionMaybes
                           [comparePartialities (domPT pt1) (domPT pt2),
                            comparePartialities (codPT pt1) (codPT pt2),
                            typeRestrInducedByPartTerm pt1,
                            typeRestrInducedByPartTerm pt2]
      pt1 :>=/==: pt2   -> collectTypeRestrictionMaybes
                           [comparePartialities (domPT pt1) (domPT pt2),
                            comparePartialities (codPT pt1) (codPT pt2),
                            typeRestrInducedByPartTerm pt1,
                            typeRestrInducedByPartTerm pt2]
      StrictPartContained pt1 pt2 -> collectTypeRestrictionMaybes
                           [comparePartialities (domPT pt1) (domPT pt2),
                            comparePartialities (codPT pt1) (codPT pt2),
                            typeRestrInducedByPartTerm pt1,
                            typeRestrInducedByPartTerm pt2]


typeRestrInducedByFormula f =
   case f of
      FV       fv    -> Just []
      OF       ofo   -> let ObjEqual o1 o2 = ofo
                        in  compareObjects o1 o2
      EF       ef    -> typeRestrInducedByElemForm ef
      VF       vf    -> typeRestrInducedByVectForm vf
      RF       rf    -> typeRestrInducedByRelaForm rf
      --PF       pf    -> typeRestrInducedByPartForm pf
      Negated  f1    -> typeRestrInducedByFormula f1
      Implies  f1 f2 -> collectTypeRestrictionMaybes
                           [typeRestrInducedByFormula f1,
                            typeRestrInducedByFormula f2]
      SemEqu   f1 f2 -> collectTypeRestrictionMaybes
                           [typeRestrInducedByFormula f1,
                            typeRestrInducedByFormula f2]
      Disjunct f1 f2 -> collectTypeRestrictionMaybes
                           [typeRestrInducedByFormula f1,
                            typeRestrInducedByFormula f2]
      Conjunct f1 f2 -> collectTypeRestrictionMaybes
                           [typeRestrInducedByFormula f1,
                            typeRestrInducedByFormula f2]

typeRestrInducedByFormulae fs = foldr1 (\x y -> collectTypeRestrictionMaybes [x,y])
                        (map typeRestrInducedByFormula fs)
```

### 4.4.2 Unification

In a transformation environment one is usually interested in a most general typing, which may be reached in building terms first with ever new domains and codomains and afterwards unifying these. The algorithm is just an implementation of the article of Krysztof Apt in the Handbook of Theoretical Computer Science, vol. B, [Apt90], so it doesn't need additional comments.

```
unifyCatObjPairsAPT :: [(CatObject,CatObject)] -> Maybe [(CatObject,CatObject)]
unifyCatObjPairsAPT []  = Just []
unifyCatObjPairsAPT ((a,b):t) =
   case a of
      OC oca -> case b of
                  OC ocb -> case oca == ocb of
                              True  -> unifyCatObjPairsAPT t
                              False -> Nothing
                  OV _   -> unifyCatObjPairsAPT ((b,a):t)
                  _      -> Nothing

      DirPro li re -> case b of
                        DirPro li' re' -> unifyCatObjPairsAPT
                                               ((li,li'):((re,re'):t))
                        OV _           -> unifyCatObjPairsAPT ((b,a):t)
                        _              -> Nothing
      DirSum li re -> case b of
                        DirSum li' re' -> unifyCatObjPairsAPT
                                               ((li,li'):((re,re'):t))
                        OV _           -> unifyCatObjPairsAPT ((b,a):t)
                        _              -> Nothing
      DirPow li    -> case b of
                        DirPow li'     -> unifyCatObjPairsAPT ((li,li'):t)
                        OV _           -> unifyCatObjPairsAPT ((b,a):t)
                        _              -> Nothing
      UnitOb       -> case b of
                        UnitOb         -> unifyCatObjPairsAPT t
                        OV _           -> unifyCatObjPairsAPT ((b,a):t)
                        _              -> Nothing
      -- QuotMod
      -- InjFrom
      -- Strict


      OV ova ->
         case b of
            OV ovb ->
             case ova == ovb of
                True  -> unifyCatObjPairsAPT t
                False ->
                 let (varsInB ,_,_,_,_,_,_,_,_,_) = syntMatUsedInCatObj  b
                     varOccursInB = ova `elem` varsInB
                 in  case varOccursInB of
                      True  -> Nothing
                      False -> let rest = unifyCatObjPairsAPT
                                    (map (\(x,y) ->
                                     (imposeTypeRestrsOnCatObj [(a,b)] x,
                                      imposeTypeRestrsOnCatObj [(a,b)] y)) t)
                               in  case rest of
                                     Just pp -> Just ((a,b) : pp)
```

```
                                Nothing -> Nothing
          _ -> let (varsInB ,_,_,_,_,_,_,_,_,_) = syntMatUsedInCatObj  b
                   varOccursInB = ova `elem` varsInB
               in  case varOccursInB of
                      True  -> Nothing
                      False -> let rest = unifyCatObjPairsAPT (map (\(x,y) ->
                                             (imposeTypeRestrsOnCatObj [(a,b)] x,
                                              imposeTypeRestrsOnCatObj
                                                      [(a,b)] y)) t)
                                   in  case rest of
                                          Just pp -> Just ((a,b) : pp)
                                          Nothing -> Nothing
```

### 4.4.3 Imposing Type Restrictions

Once one has found all the type restrictions necessary to have the terms and formulae well-formed, and has unified them, one will wish to impose the resulting substitutions. Thereby the not yet well-formed formulae will be typed in the most general form.

```
imposeONETypeRestrOnCatObj :: (CatObject,CatObject) -> CatObject -> CatObject
imposeONETypeRestrOnCatObj tr co =
   case co of
      OC _          -> co
      OV cov        -> if (\(OV v,_) -> v) tr == cov then snd $ tr else co
      DirPro o1 o2 -> DirPro  (imposeONETypeRestrOnCatObj tr o1)
                              (imposeONETypeRestrOnCatObj tr o2)
      DirSum o1 o2 -> DirSum  (imposeONETypeRestrOnCatObj tr o1)
                              (imposeONETypeRestrOnCatObj tr o2)
      DirPow o1    -> DirPow  (imposeONETypeRestrOnCatObj tr o1)
      UnitOb       -> UnitOb
      QuotMod   rt -> QuotMod (imposeONETypeRestrOnRelaTerm tr rt)
      InjFrom   vt -> InjFrom (imposeONETypeRestrOnVectTerm tr vt)
      Strict    po -> Strict  (imposeONETypeRestrOnParObj   tr po)


imposeTypeRestrsOnCatObj []    co = co
imposeTypeRestrsOnCatObj (h:t) co = imposeTypeRestrsOnCatObj t
                                    (imposeONETypeRestrOnCatObj h co)


imposeONETypeRestrOnParObj :: (CatObject,CatObject) -> ParObject -> ParObject
imposeONETypeRestrOnParObj tr po =
   case po of
      ParObj co      -> ParObj (imposeONETypeRestrOnCatObj tr co)
      ParPro po1 po2 -> ParPro (imposeONETypeRestrOnParObj tr po1)
                               (imposeONETypeRestrOnParObj tr po2)
      ParSum po1 po2 -> ParSum (imposeONETypeRestrOnParObj tr po1)
                               (imposeONETypeRestrOnParObj tr po2)
      ParPow po1     -> ParPow (imposeONETypeRestrOnParObj tr po1)
```

```
imposeTypeRestrsOnParObj []    po = po
imposeTypeRestrsOnParObj (h:t) po = imposeTypeRestrsOnParObj t
                                    (imposeONETypeRestrOnParObj h po)

imposeONETypeRestrOnElemConst :: (CatObject,CatObject) -> ElemConst -> ElemConst
imposeONETypeRestrOnElemConst tr ec =
    case ec of
       Elem s co -> Elem s (imposeONETypeRestrOnCatObj tr co)
imposeTypeRestrsOnElemConst []    ec = ec
imposeTypeRestrsOnElemConst (h:t) ec = imposeTypeRestrsOnElemConst t
                                       (imposeONETypeRestrOnElemConst h ec)

imposeONETypeRestrOnElemVari :: (CatObject,CatObject) -> ElemVari -> ElemVari
imposeONETypeRestrOnElemVari tr ev =
    case ev of
       VarE          s co -> VarE        s   (imposeONETypeRestrOnCatObj tr co)
       IndexedVarE s i co -> IndexedVarE s i (imposeONETypeRestrOnCatObj tr co)
imposeTypeRestrsOnElemVari []    ev = ev
imposeTypeRestrsOnElemVari (h:t) ev = imposeTypeRestrsOnElemVari t
                                      (imposeONETypeRestrOnElemVari h ev)


imposeONETypeRestrOnElemTerm :: (CatObject,CatObject) -> ElemTerm -> ElemTerm
imposeONETypeRestrOnElemTerm tr et =
   case et of
      EC    _  -> et
      EV    ev -> EV (imposeONETypeRestrOnElemVari tr ev)
      Pair  et1 et2 -> Pair (imposeONETypeRestrOnElemTerm tr et1)
                            (imposeONETypeRestrOnElemTerm tr et2)
      Inj1 et  o    -> Inj1 (imposeONETypeRestrOnElemTerm tr et)
                            (imposeONETypeRestrOnCatObj   tr o)
      Inj2 o   et   -> Inj2 (imposeONETypeRestrOnCatObj   tr o)
                            (imposeONETypeRestrOnElemTerm tr et)
      ThatV vt -> ThatV (imposeONETypeRestrOnVectTerm tr vt)
      SomeV vt -> SomeV (imposeONETypeRestrOnVectTerm tr vt)
      ThatR rt -> ThatR (imposeONETypeRestrOnRelaTerm tr rt)
      SomeR rt -> SomeR (imposeONETypeRestrOnRelaTerm tr rt)
      FuncAppl fc et -> FuncAppl fc(imposeONETypeRestrOnElemTerm tr et)
      VectToElem _ -> imposeONETypeRestrOnElemTerm tr $ expandDefinedElemTerm et
imposeTypeRestrsOnElemTerm []    ev = ev
imposeTypeRestrsOnElemTerm (h:t) ev = imposeTypeRestrsOnElemTerm t
                                      (imposeONETypeRestrOnElemTerm h ev)

imposeONETypeRestrOnVectConst :: (CatObject,CatObject) -> VectConst -> VectConst
imposeONETypeRestrOnVectConst tr vc =
    case vc of
       Vect s co -> Vect s (imposeONETypeRestrOnCatObj tr co)
imposeTypeRestrsOnVectConst []    vc = vc
imposeTypeRestrsOnVectConst (h:t) vc = imposeTypeRestrsOnVectConst t
                                       (imposeONETypeRestrOnVectConst h vc)
```

```
imposeONETypeRestrOnVectVari :: (CatObject,CatObject) -> VectVari -> VectVari
imposeONETypeRestrOnVectVari tr vv =
    case vv of
       VarV          s   co -> VarV          s   (imposeONETypeRestrOnCatObj tr co)
       IndexedVarV s i co -> IndexedVarV s i (imposeONETypeRestrOnCatObj tr co)

imposeTypeRestrsOnVectVari []    vv = vv
imposeTypeRestrsOnVectVari (h:t) vv = imposeTypeRestrsOnVectVari t
                                      (imposeONETypeRestrOnVectVari h vv)


imposeONETypeRestrOnVectTerm :: (CatObject,CatObject) -> VectTerm -> VectTerm
imposeONETypeRestrOnVectTerm tr vt =
   case vt of
      VC (Vect s o)  -> VC (Vect s (imposeONETypeRestrOnCatObj tr o))
      VV vv          -> VV (imposeONETypeRestrOnVectVari tr vv)
      rt1 :****: vt2 -> (imposeONETypeRestrOnRelaTerm tr rt1) :****:
                        (imposeONETypeRestrOnVectTerm tr vt2)
      vt1 :||||: vt2 -> (imposeONETypeRestrOnVectTerm tr vt1) :||||:
                        (imposeONETypeRestrOnVectTerm tr vt2)
      vt1 :&&&&: vt2 -> (imposeONETypeRestrOnVectTerm tr vt1) :&&&&:
                        (imposeONETypeRestrOnVectTerm tr vt2)
      Syq rt1 vt2    -> Syq (imposeONETypeRestrOnRelaTerm tr rt1)
                            (imposeONETypeRestrOnVectTerm tr vt2)
      NegaV      vt1 -> NegaV     (imposeONETypeRestrOnVectTerm tr vt1)
      NullV      o   -> NullV     (imposeONETypeRestrOnCatObj   tr o)
      UnivV      o   -> UnivV     (imposeONETypeRestrOnCatObj   tr o)
      SupVect    vs  -> SupVect   (imposeONETypeRestrOnVectSET  tr vs)
      InfVect    vs  -> InfVect   (imposeONETypeRestrOnVectSET  tr vs)
      PointVect  et  -> PointVect (imposeONETypeRestrOnElemTerm tr et)
      PowElemToVect et -> PowElemToVect (imposeONETypeRestrOnElemTerm tr et)
      RelaToVect rt  -> RelaToVect(imposeONETypeRestrOnRelaTerm tr rt)
      VFctAppl vf vt2 -> VFctAppl (imposeONETypeRestrOnVectFct  tr vf)
                                  (imposeONETypeRestrOnVectTerm tr vt2)
imposeTypeRestrsOnVectTerm []    vv = vv
imposeTypeRestrsOnVectTerm (h:t) vv = imposeTypeRestrsOnVectTerm t
                                      (imposeONETypeRestrOnVectTerm h vv)

imposeONETypeRestrOnRelaConst :: (CatObject,CatObject) -> RelaConst -> RelaConst
imposeONETypeRestrOnRelaConst tr rc =
    case rc of
       Rela s co co' -> Rela s (imposeONETypeRestrOnCatObj tr co)
                               (imposeONETypeRestrOnCatObj tr co')
imposeTypeRestrsOnRelaConst []    rc = rc
imposeTypeRestrsOnRelaConst (h:t) rc = imposeTypeRestrsOnRelaConst t
                                       (imposeONETypeRestrOnRelaConst h rc)

imposeONETypeRestrOnFuncConst :: (CatObject,CatObject) -> FuncConst -> FuncConst
imposeONETypeRestrOnFuncConst tr fc =
```

```
    case fc of
       Func s co co' -> Func s (imposeONETypeRestrOnCatObj tr co)
                               (imposeONETypeRestrOnCatObj tr co')
imposeTypeRestrsOnFuncConst []    fc = fc
imposeTypeRestrsOnFuncConst (h:t) fc = imposeTypeRestrsOnFuncConst t
                                        (imposeONETypeRestrOnFuncConst h fc)

imposeONETypeRestrOnRelaVari :: (CatObject,CatObject) -> RelaVari -> RelaVari
imposeONETypeRestrOnRelaVari tr rv =
   case rv of
      VarR          s   co co' -> VarR s
                                    (imposeONETypeRestrOnCatObj tr co)
                                    (imposeONETypeRestrOnCatObj tr co')
      IndexedVarR s i co co' -> IndexedVarR s i
                                    (imposeONETypeRestrOnCatObj tr co)
                                    (imposeONETypeRestrOnCatObj tr co')
imposeTypeRestrsOnRelaVari []    rv = rv
imposeTypeRestrsOnRelaVari (h:t) rv = imposeTypeRestrsOnRelaVari t
                                       (imposeONETypeRestrOnRelaVari h rv)


imposeONETypeRestrOnRelaTerm :: (CatObject,CatObject) -> RelaTerm -> RelaTerm
imposeONETypeRestrOnRelaTerm tr rt =
   case rt of
      RC (Rela s o o') -> RC (Rela s (imposeONETypeRestrOnCatObj tr o )
                                     (imposeONETypeRestrOnCatObj tr o'))
      RV rv            -> RV (imposeONETypeRestrOnRelaVari tr rv)
      rt1 :***:  rt2   -> (imposeONETypeRestrOnRelaTerm tr rt1) :***:
                          (imposeONETypeRestrOnRelaTerm tr rt2)
      rt1 :|||:  rt2   -> (imposeONETypeRestrOnRelaTerm tr rt1) :|||:
                          (imposeONETypeRestrOnRelaTerm tr rt2)
      rt1 :&&&:  rt2   -> (imposeONETypeRestrOnRelaTerm tr rt1) :&&&:
                          (imposeONETypeRestrOnRelaTerm tr rt2)
      NegaR      rt1   -> NegaR (imposeONETypeRestrOnRelaTerm tr rt1)
      Ident      o     -> Ident (imposeONETypeRestrOnCatObj tr o )
      NullR      o o'  -> NullR (imposeONETypeRestrOnCatObj tr o )
                                (imposeONETypeRestrOnCatObj tr o')
      UnivR      o o'  -> UnivR (imposeONETypeRestrOnCatObj tr o )
                                (imposeONETypeRestrOnCatObj tr o')
      Convs      rt1   -> Convs (imposeONETypeRestrOnRelaTerm tr rt1)
      vt :||--:  vt'   -> (imposeONETypeRestrOnVectTerm tr vt ) :||--:
                          (imposeONETypeRestrOnVectTerm tr vt')
      SupRela    rs    -> SupRela (imposeONETypeRestrOnRelaSET  tr rs)
      InfRela    rs    -> InfRela (imposeONETypeRestrOnRelaSET  tr rs)
      Pi         o o'  -> Pi    (imposeONETypeRestrOnCatObj tr o)
                                (imposeONETypeRestrOnCatObj tr o')
      Rho        o o'  -> Rho   (imposeONETypeRestrOnCatObj tr o)
                                (imposeONETypeRestrOnCatObj tr o')
      (:*:)      _ _   -> imposeONETypeRestrOnRelaTerm tr $ expandDefinedRelaTerm rt
      (:\/:)     _ _   -> imposeONETypeRestrOnRelaTerm tr $ expandDefinedRelaTerm rt
```

```
      SyQ        _ _   -> imposeONETypeRestrOnRelaTerm tr $ expandDefinedRelaTerm rt
      Iota       o o'  -> Iota  (imposeONETypeRestrOnCatObj tr o)
                                (imposeONETypeRestrOnCatObj tr o')
      Kappa      o o'  -> Kappa (imposeONETypeRestrOnCatObj tr o)
                                (imposeONETypeRestrOnCatObj tr o')
      CASE   rt1 rt2   -> CASE  (imposeONETypeRestrOnRelaTerm tr rt1)
                                (imposeONETypeRestrOnRelaTerm tr rt2)
      Epsi       o     -> Epsi  (imposeONETypeRestrOnCatObj tr o)
      PointDiag  et    -> PointDiag (imposeONETypeRestrOnElemTerm tr et)
      InjTerm    vt    -> InjTerm   (imposeONETypeRestrOnVectTerm tr vt)
      --Wait       rt1   -> Wait    (imposeONETypeRestrOnPartTerm tr rt1)
      Belly pt         -> Belly (imposeONETypeRestrOnPartTerm tr pt)
      ProdVectToRela _ -> imposeONETypeRestrOnRelaTerm tr $ expandDefinedRelaTerm rt
      PartOrd po       -> PartOrd $ imposeONETypeRestrOnParObj tr po
      RFctAppl rf rt2  -> RFctAppl  (imposeONETypeRestrOnRelaFct tr rf)
                                    (imposeONETypeRestrOnEVRTerm tr rt2)
imposeONETypeRestrOnEVRTerm :: (CatObject,CatObject) -> ArgEVR -> ArgEVR
imposeONETypeRestrOnEVRTerm tr evr =
   case evr of
      ArgE ae -> ArgE $ imposeONETypeRestrOnElemTerm tr ae
      ArgV av -> ArgV $ imposeONETypeRestrOnVectTerm tr av
      ArgR ar -> ArgR $ imposeONETypeRestrOnRelaTerm tr ar

imposeTypeRestrsOnRelaTerm []    rt = rt
imposeTypeRestrsOnRelaTerm (h:t) rt = imposeTypeRestrsOnRelaTerm t
                                       (imposeONETypeRestrOnRelaTerm h rt)
imposeTypeRestrsOnRelaTerms l rts = map (imposeTypeRestrsOnRelaTerm l) rts

imposeONETypeRestrOnPartTerm :: (CatObject,CatObject) -> PartTerm -> PartTerm
imposeONETypeRestrOnPartTerm tr rt =
   case rt of
      Lift       rt1  -> Lift (imposeONETypeRestrOnRelaTerm tr rt1)
      Fetus po1 rt1 po2 -> Fetus (imposeONETypeRestrOnParObj   tr po1)
                                 (imposeONETypeRestrOnRelaTerm tr rt1)
                                 (imposeONETypeRestrOnParObj   tr po2)
      pt1 :*****: pt2 -> (imposeONETypeRestrOnPartTerm tr pt1) :*****:
                         (imposeONETypeRestrOnPartTerm tr pt2)
      pt1 :|||||: pt2 -> (imposeONETypeRestrOnPartTerm tr pt1) :|||||:
                         (imposeONETypeRestrOnPartTerm tr pt2)
      pt1 :&&&&&: pt2 -> (imposeONETypeRestrOnPartTerm tr pt1) :&&&&&:
                         (imposeONETypeRestrOnPartTerm tr pt2)
      NegaPart pt1    -> NegaPart (imposeONETypeRestrOnPartTerm tr pt1)
      IdentP  po      -> IdentP   (imposeONETypeRestrOnParObj   tr po)
      NullP   po1 po2 -> NullP (imposeONETypeRestrOnParObj tr po1)
                               (imposeONETypeRestrOnParObj tr po2)
      UnivP   po1 po2 -> UnivP (imposeONETypeRestrOnParObj tr po1)
                               (imposeONETypeRestrOnParObj tr po2)
      TranspP pt1     -> TranspP (imposeONETypeRestrOnPartTerm tr pt1)
      PPi     po1 po2 -> PPi     (imposeONETypeRestrOnParObj tr po1)
                                 (imposeONETypeRestrOnParObj tr po2)
```

```
      PRho    po1 po2 -> PRho    (imposeONETypeRestrOnParObj tr po1)
                                 (imposeONETypeRestrOnParObj tr po2)
      pt1 :#: pt2     -> (imposeONETypeRestrOnPartTerm tr pt1) :#:
                         (imposeONETypeRestrOnPartTerm tr pt2)
      pt1 :\\//: pt2  -> (imposeONETypeRestrOnPartTerm tr pt1) :\\//:
                         (imposeONETypeRestrOnPartTerm tr pt2)
      PIota   po1 po2 -> PIota  (imposeONETypeRestrOnParObj tr po1)
                                (imposeONETypeRestrOnParObj tr po2)
      PKappa  po1 po2 -> PKappa (imposeONETypeRestrOnParObj tr po1)
                                (imposeONETypeRestrOnParObj tr po2)
      PEpsi   po      -> PEpsi  (imposeONETypeRestrOnParObj tr po)
imposeTypeRestrsOnPartTerm []    rt = rt
imposeTypeRestrsOnPartTerm (h:t) rt = imposeTypeRestrsOnPartTerm t
                                            (imposeONETypeRestrOnPartTerm h rt)


imposeONETypeRestrOnVectFct :: (CatObject,CatObject) -> VectFct -> VectFct
imposeONETypeRestrOnVectFct tr (VFCT vv vt) =
   VFCT (imposeONETypeRestrOnVectVari tr vv)
        (imposeONETypeRestrOnVectTerm tr vt)
imposeTypeRestrsOnVectFct []    vf = vf
imposeTypeRestrsOnVectFct (h:t) vf = imposeTypeRestrsOnVectFct   t
                                     (imposeONETypeRestrOnVectFct h vf)

imposeONETypeRestrOnRelaFct :: (CatObject,CatObject) -> RelaFct -> RelaFct
imposeONETypeRestrOnRelaFct tr (RFCT evr rt) =
   case evr of
      EVar ea -> RFCT (EVar $ imposeONETypeRestrOnElemVari tr ea)
                      (imposeONETypeRestrOnRelaTerm tr rt)
      VVar va -> RFCT (VVar $ imposeONETypeRestrOnVectVari tr va)
                      (imposeONETypeRestrOnRelaTerm tr rt)
      RVar ra -> RFCT (RVar $ imposeONETypeRestrOnRelaVari tr ra)
                      (imposeONETypeRestrOnRelaTerm tr rt)
imposeTypeRestrsOnRelaFct []    rf = rf
imposeTypeRestrsOnRelaFct (h:t) rf = imposeTypeRestrsOnRelaFct t
                                     (imposeONETypeRestrOnRelaFct h rf)

imposeONETypeRestrOnVectSET :: (CatObject,CatObject) -> VectSET -> VectSET
imposeONETypeRestrOnVectSET tr vs =
   case vs of
      VarVS s o -> VarVS s (imposeONETypeRestrOnCatObj tr o)
      VS vv  fs -> VS (imposeONETypeRestrOnVectVari tr vv)
                      (imposeONETypeRestrOnFormulae tr fs)
      VX vts co -> VX (map (imposeONETypeRestrOnVectTerm tr) vts)
                      (imposeONETypeRestrOnCatObj tr co)
imposeTypeRestrsOnVectSET []    vs = vs
imposeTypeRestrsOnVectSET (h:t) vs = imposeTypeRestrsOnVectSET t
                                     (imposeONETypeRestrOnVectSET h vs)


imposeONETypeRestrOnRelaSET :: (CatObject,CatObject) -> RelaSET -> RelaSET
```

```
imposeONETypeRestrOnRelaSET tr rs =
   case rs of
      VarRS s o o' -> VarRS s (imposeONETypeRestrOnCatObj tr o)
                              (imposeONETypeRestrOnCatObj tr o')
      RS rv  fs    -> RS (imposeONETypeRestrOnRelaVari tr rv)
                          (imposeONETypeRestrOnFormulae tr fs)
      RX rts o  o' -> RX (map (imposeONETypeRestrOnRelaTerm tr) rts)
                          (imposeONETypeRestrOnCatObj tr o)
                          (imposeONETypeRestrOnCatObj tr o')
imposeTypeRestrsOnRelaSET []    rs = rs
imposeTypeRestrsOnRelaSET (h:t) rs = imposeTypeRestrsOnRelaSET t
                                       (imposeONETypeRestrOnRelaSET h rs)

imposeONETypeRestrOnCatOForm :: (CatObject,CatObject) -> CatOForm -> CatOForm
imposeONETypeRestrOnCatOForm tr cof@(ObjEqual o1 o2) =
   ObjEqual (imposeONETypeRestrOnCatObj tr o1) (imposeONETypeRestrOnCatObj tr o2)
imposeTypeRestrsOnCatOForm []    ff = ff
imposeTypeRestrsOnCatOForm (h:t) ff = imposeTypeRestrsOnCatOForm t
                                        (imposeONETypeRestrOnCatOForm h ff)

imposeONETypeRestrOnElemForm :: (CatObject,CatObject) -> ElemForm -> ElemForm
imposeONETypeRestrOnElemForm tr ef =
   case ef of
      Equation et et' -> Equation (imposeONETypeRestrOnElemTerm tr et)
                                  (imposeONETypeRestrOnElemTerm tr et')
      NegaEqua et et' -> NegaEqua (imposeONETypeRestrOnElemTerm tr et)
                                  (imposeONETypeRestrOnElemTerm tr et')
      QuantElemForm q ev fs -> QuantElemForm q
                                   (imposeONETypeRestrOnElemVari tr ev)
                                   (imposeONETypeRestrOnFormulae tr fs)
imposeTypeRestrsOnElemForm []    ef = ef
imposeTypeRestrsOnElemForm (h:t) ef = imposeTypeRestrsOnElemForm t
                                        (imposeONETypeRestrOnElemForm h ef)

imposeONETypeRestrOnVectForm :: (CatObject,CatObject) -> VectForm -> VectForm
imposeONETypeRestrOnVectForm tr vf =
   case vf of
      vt1 :<===: vt2  -> (imposeONETypeRestrOnVectTerm tr vt1) :<===:
                         (imposeONETypeRestrOnVectTerm tr vt2)
      vt1 :>===: vt2  -> (imposeONETypeRestrOnVectTerm tr vt1) :>===:
                         (imposeONETypeRestrOnVectTerm tr vt2)
      vt1 :====: vt2  -> (imposeONETypeRestrOnVectTerm tr vt1) :====:
                         (imposeONETypeRestrOnVectTerm tr vt2)
      vt1 :<=/=: vt2  -> (imposeONETypeRestrOnVectTerm tr vt1) :<=/=:
                         (imposeONETypeRestrOnVectTerm tr vt2)
      vt1 :>=/=: vt2  -> (imposeONETypeRestrOnVectTerm tr vt1) :>=/=:
                         (imposeONETypeRestrOnVectTerm tr vt2)
      vt1 :==/=: vt2  -> (imposeONETypeRestrOnVectTerm tr vt1) :==/=:
                         (imposeONETypeRestrOnVectTerm tr vt2)
      VE  vt et       -> VE  (imposeONETypeRestrOnVectTerm tr vt)
```

```
                              (imposeONETypeRestrOnElemTerm tr et)
      VectInSet vt vs -> VectInSet (imposeONETypeRestrOnVectTerm tr vt)
                                   (imposeONETypeRestrOnVectSET  tr vs)
      QuantVectForm q vv fs -> QuantVectForm q
                                   (imposeONETypeRestrOnVectVari tr vv)
                                   (imposeONETypeRestrOnFormulae tr fs)
imposeTypeRestrsOnVectForm []    vf = vf
imposeTypeRestrsOnVectForm (h:t) vf = imposeTypeRestrsOnVectForm t
                                        (imposeONETypeRestrOnVectForm h vf)


imposeONETypeRestrOnRelaForm :: (CatObject,CatObject) -> RelaForm -> RelaForm
imposeONETypeRestrOnRelaForm tr rf =
   case rf of
      rt1 :<==: rt2   -> (imposeONETypeRestrOnRelaTerm tr rt1) :<==:
                         (imposeONETypeRestrOnRelaTerm tr rt2)
      rt1 :>==: rt2   -> (imposeONETypeRestrOnRelaTerm tr rt1) :>==:
                         (imposeONETypeRestrOnRelaTerm tr rt2)
      rt1 :===: rt2   -> (imposeONETypeRestrOnRelaTerm tr rt1) :===:
                         (imposeONETypeRestrOnRelaTerm tr rt2)
      rt1 :<=/: rt2   -> (imposeONETypeRestrOnRelaTerm tr rt1) :<=/:
                         (imposeONETypeRestrOnRelaTerm tr rt2)
      rt1 :>=/: rt2   -> (imposeONETypeRestrOnRelaTerm tr rt1) :>=/:
                         (imposeONETypeRestrOnRelaTerm tr rt2)
      rt1 :=/=: rt2   -> (imposeONETypeRestrOnRelaTerm tr rt1) :=/=:
                         (imposeONETypeRestrOnRelaTerm tr rt2)
      REE rt et1 et2  -> REE (imposeONETypeRestrOnRelaTerm tr rt)
                             (imposeONETypeRestrOnElemTerm tr et1)
                             (imposeONETypeRestrOnElemTerm tr et2)
      RelaInSet rt rs -> RelaInSet (imposeONETypeRestrOnRelaTerm tr rt)
                                   (imposeONETypeRestrOnRelaSET  tr rs)
      QuantRelaForm q rv fs -> QuantRelaForm q
                                   (imposeONETypeRestrOnRelaVari tr rv)
                                   (imposeONETypeRestrOnFormulae tr fs)
imposeTypeRestrsOnRelaForm []    rf = rf
imposeTypeRestrsOnRelaForm (h:t) rf = imposeTypeRestrsOnRelaForm t
                                        (imposeONETypeRestrOnRelaForm h rf)
imposeTypeRestrsOnRelaForms trs rfs =
   map (imposeTypeRestrsOnRelaForm trs) rfs

imposeONETypeRestrOnPartForm :: (CatObject,CatObject) -> PartForm -> PartForm
imposeONETypeRestrOnPartForm tr pf =
   case pf of
      pt1 :<====: pt2 -> (imposeONETypeRestrOnPartTerm tr pt1) :<====:
                         (imposeONETypeRestrOnPartTerm tr pt2)
      pt1 :>====: pt2 -> (imposeONETypeRestrOnPartTerm tr pt1) :>====:
                         (imposeONETypeRestrOnPartTerm tr pt2)
      pt1 :<=/==: pt2 -> (imposeONETypeRestrOnPartTerm tr pt1) :<=/==:
                         (imposeONETypeRestrOnPartTerm tr pt2)
      pt1 :>=/==: pt2 -> (imposeONETypeRestrOnPartTerm tr pt1) :>=/==:
```

```
                        (imposeONETypeRestrOnPartTerm tr pt2)
      StrictPartContained pt1 pt2 -> StrictPartContained
                        (imposeONETypeRestrOnPartTerm tr pt1)
                        (imposeONETypeRestrOnPartTerm tr pt2)
imposeTypeRestrsOnPartForm []    pf = pf
imposeTypeRestrsOnPartForm (h:t) pf = imposeTypeRestrsOnPartForm t
                                          (imposeONETypeRestrOnPartForm h pf)


imposeONETypeRestrOnFormula :: (CatObject,CatObject) -> Formula -> Formula
imposeONETypeRestrOnFormula tr f =
   case f of
      OF ff -> OF (imposeONETypeRestrOnCatOForm tr ff)
      EF ef -> EF (imposeONETypeRestrOnElemForm tr ef)
      VF vf -> VF (imposeONETypeRestrOnVectForm tr vf)
      RF rf -> RF (imposeONETypeRestrOnRelaForm tr rf)
      --PF pf -> PF (imposeONETypeRestrOnPartForm tr pf)
      Verum          -> Verum
      Falsum         -> Falsum
      Negated  f1    -> Negated  (imposeONETypeRestrOnFormula tr f1)
      Implies  f1 f2 -> Implies  (imposeONETypeRestrOnFormula tr f1)
                                 (imposeONETypeRestrOnFormula tr f2)
      SemEqu   f1 f2 -> SemEqu   (imposeONETypeRestrOnFormula tr f1)
                                 (imposeONETypeRestrOnFormula tr f2)
      Disjunct f1 f2 -> Disjunct (imposeONETypeRestrOnFormula tr f1)
                                 (imposeONETypeRestrOnFormula tr f2)
      Conjunct f1 f2 -> Conjunct (imposeONETypeRestrOnFormula tr f1)
                                 (imposeONETypeRestrOnFormula tr f2)
imposeTypeRestrsOnFormula []    f = f
imposeTypeRestrsOnFormula (h:t) f = imposeTypeRestrsOnFormula t
                                    (imposeONETypeRestrOnFormula h f)

imposeONETypeRestrOnFormulae :: (CatObject,CatObject) -> [Formula] -> [Formula]
imposeONETypeRestrOnFormulae tr fs =
   map (imposeONETypeRestrOnFormula tr) fs
imposeTypeRestrsOnFormulae []    f = f
imposeTypeRestrsOnFormulae (h:t) f = imposeTypeRestrsOnFormulae t
                                     (imposeONETypeRestrOnFormulae h f)
```

### 4.4.4 Obtaining Most General Typings

Our approach for writing down a rule will later be as follows. In the course of writing, we do not check for well-formedness at every stage. Once the terms are written in total, however, we look for the necessary restrictions induced by the terms, or formulae, respectively. Only these shall afterwards be imposed to the formulae involved. This guarantees the most general typing to the rules.

```
generalType collectFct imposeFct t =
   let tyRe = collectFct t
```

```
       tyReUnif = case tyRe of
                     Just  x -> unifyCatObjPairsAPT x
                     Nothing -> Nothing
   in  case tyReUnif of
          Just  x -> imposeFct x t
          Nothing -> t

generalTypeOfVectTerm vt   = generalType
   typeRestrInducedByVectTerm    imposeTypeRestrsOnVectTerm  vt
generalTypeOfRelaTerm rt   = generalType
   typeRestrInducedByRelaTerm    imposeTypeRestrsOnRelaTerm  rt
generalTypeOfRelaTerms rts = generalType
   typeRestrInducedByRelaTerms   imposeTypeRestrsOnRelaTerms rts
generalTypeOfRelaFct rfct  = generalType
   typeRestrInducedByRelaFct     imposeTypeRestrsOnRelaFct   rfct
generalTypeOfRelaSET rs    = generalType
   typeRestrInducedByRelaSet     imposeTypeRestrsOnRelaSET   rs
generalTypeOfElemForm ef   = generalType
   typeRestrInducedByElemForm    imposeTypeRestrsOnElemForm  ef
generalTypeOfVectForm vf   = generalType
   typeRestrInducedByVectForm    imposeTypeRestrsOnVectForm  vf
generalTypeOfRelaForm rf   = generalType
   typeRestrInducedByRelaForm    imposeTypeRestrsOnRelaForm  rf
generalTypeOfRelaForms rfs = generalType
   typeRestrInducedByRelaForms   imposeTypeRestrsOnRelaForms rfs

generalTypeOfFormula f =
   case f of
      -- OF
      EF ef -> EF (generalTypeOfElemForm ef)
      VF vf -> VF (generalTypeOfVectForm vf)
      RF rf -> RF (generalTypeOfRelaForm rf)
      --PF       pf    ->
      Negated  f1    -> Negated (generalTypeOfFormula f1)
      Implies  f1 f2 -> let [f3,f4] = generalTypeOfFormulae [f1,f2]
                        in  Implies f3 f4
      SemEqu   f1 f2 -> let [f3,f4] = generalTypeOfFormulae [f1,f2]
                        in  SemEqu f3 f4
      Disjunct f1 f2 -> let [f3,f4] = generalTypeOfFormulae [f1,f2]
                        in  Disjunct f3 f4
      Conjunct f1 f2 -> let [f3,f4] = generalTypeOfFormulae [f1,f2]
                        in  Conjunct f3 f4
```

It should be noticed that one must not form the general types separately as one might easily be tempted to, implementing the latter as

`SemEqu f1 f2 -> SemEqu (generalTypeOfFormula f1) (generalTypeOfFormula f2)`

This would probably result in an erroneous typing. According to the different construction of `f1, f2` in the Schröder equivalences, e.g., the algorithm might decide to substitute differently in the two formulae.

```
generalTypeOfFormulae fs = let tyRe = typeRestrInducedByFormulae fs
                               tyReUnif = case tyRe of
                                            Just x  -> unifyCatObjPairsAPT x
                                            Nothing -> Nothing
                           in  case tyReUnif of
                                  Just  x -> imposeTypeRestrsOnFormulae x fs
                                  Nothing -> fs
generalTypeOfTheory th =
   let TH ss ocs ecs vcs rcs fcs vfs rfs fs = th
       tyRe = typeRestrInducedByFormulae fs
       tyReUnif = case tyRe of
                    Just x  -> unifyCatObjPairsAPT x
                    Nothing -> Nothing
   in  case tyReUnif of
         Just  x -> TH ss
                       (map (imposeTypeRestrsOnCatObj    x) ocs)
                       (map (imposeTypeRestrsOnElemConst x) ecs)
                       (map (imposeTypeRestrsOnVectConst x) vcs)
                       (map (imposeTypeRestrsOnRelaConst x) rcs)
                       (map (imposeTypeRestrsOnFuncConst x) fcs)
                       (map (imposeTypeRestrsOnVectFct   x) vfs)
                       (map (imposeTypeRestrsOnRelaFct   x) rfs)
                       (     imposeTypeRestrsOnFormulae  x  fs )
         Nothing -> th
```

## 4.5 Formula Translation

Once term and/or formulae are built, one will immediately start transforming them in one way or the other. A main example is transformation within some proof system. But also transformation to TEX-text or ASCII-text is some sort of a transformation. If in an environment first-order formulae are supposed to be provided, one often feels that it is rather cumbersome and error-prone to provide them. In such cases, one will often formulate in a higher relational language and afterwards translate to first-order form — again a translation. All the main translations of this kind are collected here.

### 4.5.1 Translation to First-Order Form

We provide for a translation of relation or vector 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 more complex forms.

As we have to generate variable names in the course of such a translation, we should take care, that they do not interfere with already existing ones. We therefore introduce some accounting on the variables already used.

There exists a difficult borderline between first-order logic and relational logic in the form explained here. When using vectors, we use subsets and thereby enter the realm of second-order logic. Nonetheless, these vectors are handled much in the same way as elements. So it is

interesting to observe, where the differences between first-order and second-order logic actually show up. We simply cannot translate all of our relational language into the element-oriented form. In particular, quantifications over vectors or relations cannot be formulated. Expressivity of the relational logic is, thus, above that of first-order logic.

On the other hand side, we have formulated our relational language much in the same way as a first-order language. The question might, therefore, arise as where to spot the differences. One should bear in mind, that the relational language is burdened with the existence of non-standard models. Vector or relation constants will be translated into unary or binary predicates. Vector or relation variables cannot be translated.

```
data VariUsed = VU [CatObjVar] [ElemVari] [VectVari] [RelaVari] [FormVari]
variableName (VarE s _) = s

translateVectConst :: VariUsed -> ElemVari -> VectConst -> Formula
translateVectConst _ ev vc = VF (VE (VC vc) (EV ev))

translateVectVari  :: VariUsed -> ElemVari -> VectVari  -> Formula
translateVectVari  _ ev vv = VF (VE (VV vv) (EV ev))

translateVectTerm  :: VariUsed -> ElemVari -> VectTerm  -> Formula
translateVectTerm vu ev vt =
   case vt of
      VC vc          -> translateVectConst vu ev vc
      VV vv          -> translateVectVari  vu ev vv
      rt1 :****: UnivV o ->
         let (o1,o2) = typeOfRT rt1
             VU cos evs vvs rvs fvs = vu
             av = nextElemVari o2 evs
             vu' = VU cos (av:evs) vvs rvs fvs
         in  EF (QuantElemForm Exis av
                [translateRelaTerm vu' ev  av rt1])
      rt1 :****: vt2 ->
         let (o1,o2) = typeOfRT rt1
             VU cos evs vvs rvs fvs = vu
             av = nextElemVari o2 evs
             vu' = VU cos (av:evs) vvs rvs fvs
         in  EF (QuantElemForm Exis av
              [Conjunct (translateRelaTerm vu' ev  av rt1)
                        (translateVectTerm vu'     av vt2)])
      vt1 :||||: vt2   -> Disjunct (translateVectTerm vu ev vt1)
                                   (translateVectTerm vu ev vt2)
      vt1 :&&&&: vt2   -> Conjunct (translateVectTerm vu ev vt1)
                                   (translateVectTerm vu ev vt2)
      NegaV      vt    -> Negated  (translateVectTerm vu ev vt)
      NullV      o     -> VF (VE (NullV o) (EV ev))
      UnivV      o     -> VF (VE (UnivV o) (EV ev))
      --SupVect    vs  ->
      --InfVect    vs  ->
      RelaToVect _     -> translateVectTerm vu ev $ expandDefinedVectTerm vt
```

```
      PointVect  et    -> EF (Equation (EV ev) et)
      PowElemToVect et -> EF (Equation (EV ev) et)
      Syq _ _          -> translateVectTerm vu ev $ expandDefinedVectTerm vt


translateVectTerms  :: VariUsed -> ElemVari -> [VectTerm]  -> [Formula]
translateVectTerms vu ev vts = map (translateVectTerm vu ev) vts

translateRelaConst :: VariUsed -> ElemVari -> ElemVari -> RelaConst -> Formula
translateRelaConst _ ev ev' rc = RF (REE (RC rc) (EV ev) (EV ev'))

translateRelaVari  :: VariUsed -> ElemVari -> ElemVari -> RelaVari  -> Formula
translateRelaVari  _ ev ev' rv = RF (REE (RV rv) (EV ev) (EV ev'))

translateRelaTerm  :: VariUsed -> ElemVari -> ElemVari -> RelaTerm  -> Formula
translateRelaTerm vu ev ev' rt =
   case rt of
      RC rc          -> translateRelaConst vu ev ev' rc
      RV rv          -> translateRelaVari  vu ev ev' rv
      rt1 :***: rt2  ->
         let (o1,o2) = typeOfRT rt1
             (o3,o4) = typeOfRT rt2
             VU cos evs vvs rvs fvs = vu
             av = nextElemVari o2 evs
             vu' = VU cos (av:evs) vvs rvs fvs
         in  EF (QuantElemForm Exis av
                 [Conjunct (translateRelaTerm vu' ev av  rt1)
                           (translateRelaTerm vu' av ev' rt2)])
      rt1 :|||: rt2  -> Disjunct (translateRelaTerm vu ev ev' rt1)
                                 (translateRelaTerm vu ev ev' rt2)
      rt1 :&&&: rt2  -> Conjunct (translateRelaTerm vu ev ev' rt1)
                                 (translateRelaTerm vu ev ev' rt2)
      NegaR     rt1  -> Negated $ translateRelaTerm vu ev ev' rt1
      Ident      _   -> EF (Equation (EV ev) (EV ev'))
      NullR      _ _ -> Falsum
      UnivR      _ _ -> Verum
      Convs      rt1 -> translateRelaTerm vu ev' ev rt1
      vt1 :||--: vt2 -> Conjunct (translateVectTerm vu ev  vt1)
                                 (translateVectTerm vu ev' vt2)
      --SupRela    rs  ->
      --InfRela    rs  ->
      Pi       o o'  -> RF (REE (Pi    o o') (EV ev) (EV ev'))
      Rho      o o'  -> RF (REE (Rho   o o') (EV ev) (EV ev'))
      (:*:)    _ _   -> translateRelaTerm vu ev ev' $ expandDefinedRelaTerm rt
      (:\/:)   _ _   -> translateRelaTerm vu ev ev' $ expandDefinedRelaTerm rt
      SyQ      _ _   -> translateRelaTerm vu ev ev' $ expandDefinedRelaTerm rt
      Iota     o o'  -> RF (REE (Iota  o o') (EV ev) (EV ev'))
      Kappa    o o'  -> RF (REE (Kappa o o') (EV ev) (EV ev'))
      CASE   rt1 rt2 ->
         Disjunct (Conjunct (translateVectTerm vu ev (Convs (Iota  (domRT rt1)
```

```
                                        (domRT rt2)) :****: UnivV (domRT rt1)))
                              (translateRelaTerm vu ev ev' rt1))
                    (Conjunct (translateVectTerm vu ev (Convs (Kappa (domRT rt1)
                                        (domRT rt2)) :****: UnivV (domRT rt1)))
                              (translateRelaTerm vu ev ev' rt2))
      Epsi     o     -> RF (REE (Epsi  o   ) (EV ev) (EV ev'))
      PointDiag et   -> Conjunct (EF (Equation (EV ev)  et))
                                 (EF (Equation (EV ev') et))
      InjTerm   vt   -> RF (REE (InjTerm vt) (EV ev) (EV ev'))
      ProdVectToRela _ -> translateRelaTerm vu ev ev' $ expandDefinedRelaTerm rt
      --Wait rt1   ->

translateRelaTerms :: VariUsed -> ElemVari -> ElemVari -> [RelaTerm]  -> [Formula]
translateRelaTerms vu ev ev' rts = map (translateRelaTerm vu ev ev') rts


translateVectSET :: VariUsed -> ElemVari -> VectSET -> Formula
translateVectSET vu ev vs =
   case vs of
      --VarVS s o o' ->
      VS vv  fs -> VF $ QuantVectForm Univ vv (translateFormulae vu fs)
      VX vts _  -> foldr1 (\f1 f2 -> Conjunct f1 f2) (translateVectTerms vu ev vts)


translateRelaSET :: VariUsed -> ElemVari -> ElemVari -> RelaSET -> Formula
translateRelaSET vu ev ev' rs =
   case rs of
      --VarRS s o o' ->
      RS rv  fs    -> RF $ QuantRelaForm Univ rv (translateFormulae vu fs)
      RX rts _  _  -> foldr1 (\f1 f2 -> Conjunct f1 f2)
                                              (translateRelaTerms vu ev ev' rts)

translateElemForm :: VariUsed -> ElemForm -> Formula
translateElemForm _ ef =
   case ef of
      Equation et1 et2         -> EF ef
      NegaEqua et1 et2         -> EF ef
      QuantElemForm q ev fs -> EF (QuantElemForm q ev fs)


translateVectForm :: VariUsed -> ElemVari -> VectForm -> Formula
translateVectForm vu ev vf =
   case vf of
      UnivV o :<===: vt2 ->
         let VU cos evs vvs rvs fvs = vu
             av = nextElemVari o evs
             vu' = VU cos (av:evs) vvs rvs
         in  translateVectTerm vu ev vt2
      vt1 :<===: NullV o ->
         let VU cos evs vvs rvs fvs = vu
             av = nextElemVari o evs
```

```
            vu' = VU cos (av:evs) vvs rvs
         in  translateVectTerm vu ev (NegaV vt1)
      vt1 :<===: vt2 -> Implies (translateVectTerm vu ev vt1)
                                (translateVectTerm vu ev vt2)
      _   :>===: _   -> translateFormula vu $ expandDefinedVectForm vf
      _   :====: _   -> translateFormula vu $ expandDefinedVectForm vf
      _   :<=/=: _   -> translateFormula vu $ expandDefinedVectForm vf
      _   :>=/=: _   -> translateFormula vu $ expandDefinedVectForm vf
      _   :==/=: _   -> translateFormula vu $ expandDefinedVectForm vf
      VectInSet vt vs -> let trt = translateVectTerm vu ev vt
                             trs = translateVectSET  vu ev vs
                         in  Implies trt trs
      VE  vt et      ->
         let VU cos evs vvs rvs fvs = vu
             av = nextElemVari (domVT vt) evs
             vu' = VU cos (av:evs) vvs rvs fvs
             translateWithVari = translateVectTerm vu' av vt
         in  substituteInFormulaAccordingToList
                ([],[(av, et)],[],[],[]) translateWithVari
      QuantVectForm _ _ _ -> error "Trying to quantify over a unary predicate"
```

Here, we could again see the borderline between first-order and second-order logic. Since the result is supposed to be first-order, it is not allowed to translate quantification over vectors.

```
translateRelaForm :: VariUsed -> ElemVari -> ElemVari -> RelaForm  -> Formula
translateRelaForm vu ev ev' rf =
   case rf of
      rt1 :<==: rt2 -> Implies (translateRelaTerm vu ev ev' rt1)
                               (translateRelaTerm vu ev ev' rt2)
      _   :>==: _   -> translateFormula vu $ expandDefinedRelaForm rf
      _   :===: _   -> translateFormula vu $ expandDefinedRelaForm rf
      _   :<=/: _   -> translateFormula vu $ expandDefinedRelaForm rf
      _   :>=/: _   -> translateFormula vu $ expandDefinedRelaForm rf
      _   :=/=: _   -> translateFormula vu $ expandDefinedRelaForm rf

      rt1 :>=/: rt2 -> let (o1, o2) = typeOfRT rt1
                           VU cos evs vvs rvs fvs = vu
                           av1 = nextElemVari o1 evs
                           av2 = nextElemVari o2 (av1:evs)
                           vu' = VU cos (av1:(av2:evs)) vvs rvs fvs
                       in  Negated $ EF (QuantElemForm Univ av1
                           [EF (QuantElemForm Univ av2
                              [Implies (translateRelaTerm vu' av1 av2 rt2)
                                       (translateRelaTerm vu' av1 av2 rt1)])])
      --RelaInSet          rt rs
      REE rt et1 et2 -> let (o1, o2) = typeOfRT rt
                            VU cos evs vvs rvs fvs = vu
                            av1 = nextElemVari (domRT rt) evs
                            av2 = nextElemVari (codRT rt) (av1:evs)
                            vu' = VU cos (av1:(av2:evs)) vvs rvs fvs
```

```
                          translateWithVaris = translateRelaTerm vu' av1 av2 rt
                      in  substituteInFormulaAccordingToList
                           ([],[(av1, et1),(av2, et2)],[],[],[])
                           translateWithVaris
      QuantRelaForm _ _ _ -> error "Trying to quantify over a binary predicate"
translateRelaForms :: VariUsed -> ElemVari -> ElemVari -> [RelaForm] -> [Formula]
translateRelaForms vu ev ev' rfs = map (translateRelaForm vu ev ev') rfs
```

Here again, we could see the borderline between first-order and second-order logic. Since the result is supposed to be first-order, it is not allowed to translate quantification over relations.

```
translateFormula vu f =
   case f of
      -- OF _  ->
      EF _  -> f
      VF (UnivV o  :<===: vt2) ->
         let VU cos evs vvs rvs fvs = vu
             av  = nextElemVari o evs
             et  = EV av
             vu' = VU cos (av:evs) vvs rvs fvs
         in  translateElemForm vu
             (QuantElemForm Univ av [translateVectForm vu av (VE vt2 et)])
      VF (vt1 :<===: NullV o) ->
         let VU cos evs vvs rvs fvs = vu
             av  = nextElemVari o evs
             et  = EV av
             vu' = VU cos (av:evs) vvs rvs fvs
         in  translateElemForm vu
             (QuantElemForm Univ av [translateVectForm vu av (VE (NegaV vt1) et)])
      VF (vt1 :<===: vt2) ->
         let VU cos evs vvs rvs fvs = vu
             av  = nextElemVari (domVT vt1) evs
             et  = EV av
             vu' = VU cos (av:evs) vvs rvs fvs
         in  translateElemForm vu
             (QuantElemForm Univ av [Implies
                                     (translateVectForm vu' av (VE vt1 et))
                                     (translateVectForm vu' av (VE vt2 et))])
      VF (_ :>===: _) -> translateFormula vu $ expandDefinedFormula f
      VF (_ :====: _) -> translateFormula vu $ expandDefinedFormula f
      VF (_ :<=/=: _) -> translateFormula vu $ expandDefinedFormula f
      VF (_ :>=/=: _) -> translateFormula vu $ expandDefinedFormula f
      VF (_ :==/=: _) -> translateFormula vu $ expandDefinedFormula f
      VF (VE vt tx)   -> translateVectForm vu (VarE "" (OV $ VarO "")) (VE vt tx)
      VF vf -> let o = domVF vf
                   VU cos evs vvs rvs fvs = vu
                   av = nextElemVari o evs
                   vu' = VU cos (av:evs) vvs rvs fvs
               in  EF (QuantElemForm Univ av [translateVectForm vu' av vf])
      RF (UnivR o o' :<==: rt2) ->
```

```
   let VU cos evs vvs rvs fvs = vu
       av  = nextElemVari o  evs
       av' = nextElemVari o' (av:evs)
       et  = EV av
       et' = EV av'
       vu' = VU cos (av:(av':evs)) vvs rvs fvs
   in  EF $ QuantElemForm Univ av
          [EF $ QuantElemForm Univ av'
                [translateRelaForm vu' av av' (REE rt2 et et')]]
RF (rt1 :<==: NullR o o') ->
   let VU cos evs vvs rvs fvs = vu
       av  = nextElemVari o evs
       av' = nextElemVari o' (av:evs)
       et  = EV av
       et' = EV av'
       vu' = VU cos (av:(av':evs)) vvs rvs fvs
   in  EF $ QuantElemForm Univ av
          [EF $ QuantElemForm Univ av'
                [translateRelaForm vu' av av' (REE (NegaR rt1) et et')]]
RF (Ident o :<==: rt2) ->
   let VU cos evs vvs rvs fvs = vu
       av  = nextElemVari o evs
       et  = EV av
       vu' = VU cos (av:evs) vvs rvs fvs
   in  EF $ QuantElemForm Univ av
            [translateRelaForm vu' av av (REE rt2 et et)]
RF (rt1 :<==: rt2) ->
   let VU cos evs vvs rvs fvs = vu
       av1 = nextElemVari (domRT rt1) evs
       av2 = nextElemVari (codRT rt1) (av1:evs)
       et1 = EV av1
       et2 = EV av2
       vu' = VU cos (av2:(av1:evs)) vvs rvs fvs
   in  translateElemForm vu
       (QuantElemForm Univ av1 [EF $ (QuantElemForm Univ av2
          [Implies (translateRelaForm vu' av1 av2 (REE rt1 et1 et2))
                   (translateRelaForm vu' av1 av2 (REE rt2 et1 et2))])])
RF (_ :>==: _) -> translateFormula vu $ expandDefinedFormula f
RF (_ :===: _) -> translateFormula vu $ expandDefinedFormula f
RF (_ :<=/: _) -> translateFormula vu $ expandDefinedFormula f
RF (_ :>=/: _) -> translateFormula vu $ expandDefinedFormula f
RF (_ :=/=: _) -> translateFormula vu $ expandDefinedFormula f
RF (REE rt tx ty) -> translateRelaForm vu (VarE "" (OV $ VarO ""))
                     (VarE "" (OV $ VarO "")) (REE rt tx ty)
RF rf -> let (o1, o2) = typeOfRF rf
             VU cos evs vvs rvs fvs = vu
             av1 = nextElemVari o1 evs
             av2 = nextElemVari o2 (av1:evs)
             vu' = VU cos (av1:(av2:evs)) vvs rvs fvs
         in  EF (QuantElemForm Univ av1
```

```
                        [EF (QuantElemForm Univ av2
                             [translateRelaForm vu' av1 av2 rf])])
      --PF  rf ->
      Verum          -> Verum
      Falsum         -> Falsum
      Negated  f1    -> Negated  (translateFormula vu f1)
      Implies  f1 f2 -> Implies  (translateFormula vu f1) (translateFormula vu f2)
      SemEqu   f1 f2 -> SemEqu   (translateFormula vu f1) (translateFormula vu f2)
      Disjunct f1 f2 -> Disjunct (translateFormula vu f1) (translateFormula vu f2)
      Conjunct f1 f2 -> Conjunct (translateFormula vu f1) (translateFormula vu f2)

translateFormulae :: VariUsed -> [Formula] -> [Formula]
translateFormulae vu fs = map (translateFormula vu) fs
```

### 4.5.2 Translation into TEX

To make the ugly type-carrying language more readable, we provide for a straightforward translation into TEX. In a number of cases, one may decide for a long or a short version. This is regulated by the first boolean argument being `True` for long and `False` for short.

```
makeTeXCatObject :: Bool -> CatObject -> String
makeTeXCatObject b co1 =
   case co1 of
      OC (CstO co) -> co
      OV (VarO vo) -> vo
      OV (IndexedVarO vo i) -> vo ++ "\\_{" ++ show i ++"}"
      DirPro o o'  -> makeTeXCatObject b o ++ " \\times " ++ makeTeXCatObject b o'
      DirSum o o'  -> makeTeXCatObject b o ++    " + "    ++ makeTeXCatObject b o'
      DirPow o     -> "{\\cal P}(" ++ makeTeXCatObject b o ++ ")"
      UnitOb       -> "\\hbox{1\\kern-0.12cmI}"
      QuotMod   rt -> makeTeXCatObject b (domRT rt) ++ "/" ++ "(" ++
                      (makeTeXRelaTerm b rt) ++ ")"
      InjFrom   vt -> "SubObject" ++ "(" ++ (makeTeXVectTerm  b vt) ++ ")"
      Strict    po -> "Strict"    ++ "(" ++ (makeTeXParObject b po) ++ ")"


makeTeXParObject :: Bool -> ParObject -> String
makeTeXParObject b po =
   case po of
      ParObj co     -> makeTeXCatObject b co ++ "\\sp{\\rot\\bot\\schwarz}"
      ParPro po po' -> makeTeXParObject b po ++ " \\rot\\times\\schwarz " ++
                       makeTeXParObject b po'
      ParSum po po' -> makeTeXParObject b po ++    " \\rot +\\schwarz "   ++
                       makeTeXParObject b po'
      ParPow po     -> "{\\cal\\rot P\\schwarz}(" ++
                                makeTeXParObject b po ++ ")"
```

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

makeTeXElemVari  :: Bool -> ElemVari  -> String
makeTeXElemVari b ev =
   case ev of
      VarE          s   o -> case b of
                               True  -> s ++ "_{" ++ (makeTeXCatObject b o) ++ "}"
                               False -> s
      IndexedVarE s i o -> case b of
                               True  -> "{" ++ s ++ "_{" ++ show i ++ "}}_{(" ++
                                        (makeTeXCatObject b o) ++ ")}"
                               False -> s ++ "_{" ++ show i ++ "}"

makeTeXVectConst :: Bool -> VectConst -> String
makeTeXVectConst b (Vect s o) =
   case b of
      True  -> s ++ "_{" ++ (makeTeXCatObject b o) ++ "}"
      False -> s

makeTeXVectVari  :: Bool -> VectVari  -> String
makeTeXVectVari b vv =
   case vv of
      VarV          s   o -> case b of
                               True  -> s ++ "_{" ++ (makeTeXCatObject b o) ++ "}"
                               False -> s
      IndexedVarV s i o -> case b of
                               True  -> s ++ (makeTeXCatObject b o) ++
                                        "_{" ++ show i ++ "}"
                               False -> s ++ "_{" ++ show i ++ "}"

makeTeXRelaConst :: Bool -> RelaConst -> String
makeTeXRelaConst b (Rela s o o') =
   case b of
      True  -> s ++ "_{" ++ (makeTeXCatObject b o) ++
               "," ++ (makeTeXCatObject b o') ++ "}"
      False -> s

makeTeXFuncConst :: Bool -> FuncConst -> String
makeTeXFuncConst b (Func s o o') =
   case b of
      True  -> s ++ "_{" ++ (makeTeXCatObject b o) ++
               "," ++ (makeTeXCatObject b o') ++ "}"
      False -> s



makeTeXRelaVari  :: Bool -> RelaVari  -> String
makeTeXRelaVari b rv =
```

```
   case rv of
      VarR          s o o' -> case b of
                                 True  -> s ++ "_{" ++ (makeTeXCatObject b o)  ++
                                                  "," ++ (makeTeXCatObject b o') ++ "}"
                                 False -> s
      IndexedVarR s i o o' -> case b of
                                 True  -> "{" ++ s ++ "_{" ++ show i ++ "}}" ++
                                          "_\\hbox{\\small$ " ++
                                          (makeTeXCatObject b o) ++
                                          "," ++ (makeTeXCatObject b o') ++ "$}"
                                 False -> s ++ "_{" ++ show i ++ "}"

makeTeXEVRVari  :: Bool -> EVRVari  -> String
makeTeXEVRVari b evr =
   case evr of
      EVar ev -> makeTeXElemVari b ev
      VVar vv -> makeTeXVectVari b vv
      RVar rv -> makeTeXRelaVari b rv

makeTeXElemTerm  :: Bool -> ElemTerm  -> String
makeTeXElemTerm b et =
   case et of
      EV iv -> makeTeXElemVari  b iv
      EC ic -> makeTeXElemConst b ic
      Pair  et1 et2 -> "(" ++ makeTeXElemTerm b et1 ++ "," ++
                               makeTeXElemTerm b et2 ++ ")"
      Inj1 et  o    -> "\\iota1(" ++ makeTeXElemTerm b et ++ ")"
      Inj2 o   et   -> "\\iota2(" ++ makeTeXElemTerm b et ++ ")"
      ThatV vt ->
         case b of
            True  -> "{\\tt That\\ } \\theta\\in " ++
                     (makeTeXCatObject b (domVT vt)) ++ ": " ++
                     (makeTeXVectForm b (VE vt (EV (VarE "\\theta " (domVT vt)))))
            False -> "{\\tt That\\ } \\theta : " ++
                     (makeTeXVectForm b (VE vt (EV (VarE "\\theta " (domVT vt)))))
      SomeV vt ->
         case b of
            True  -> "{\\tt Some\\ } \\theta\\in " ++
                     (makeTeXCatObject b (domVT vt)) ++ ": " ++
                     (makeTeXVectForm b (VE vt (EV (VarE "\\theta " (domVT vt)))))
            False -> "{\\tt Some\\ } \\theta : " ++
                     (makeTeXVectForm b (VE vt (EV (VarE "\\theta " (domVT vt)))))
      ThatR rt ->
         case b of
            True  -> "{\\tt That\\ } \\theta\\in " ++
                     (makeTeXCatObject b (domRT rt)) ++ ": " ++
                     (makeTeXRelaForm b (REE rt (EV (VarE "\\theta " (domRT rt)))
                                        (EV (VarE "\\theta " (domRT rt)))))
            False -> "{\\tt That\\ } \\theta : " ++
                     (makeTeXRelaForm b (REE rt (EV (VarE "\\theta " (domRT rt)))
```

```
                                    (EV (VarE "\\theta " (domRT rt)))))
      SomeR rt ->
         case b of
            True  -> "{\\tt Some\\ } \\theta\\in " ++
                     (makeTeXCatObject b (domRT rt)) ++ ": " ++
                     (makeTeXRelaForm b (REE rt (EV (VarE "\\theta " (domRT rt)))
                                    (EV (VarE "\\theta " (domRT rt)))))
            False -> "{\\tt Some\\ } \\theta : " ++
                     (makeTeXRelaForm b (REE rt (EV (VarE "\\theta " (domRT rt)))
                                    (EV (VarE "\\theta " (domRT rt)))))
      FuncAppl fc et' -> makeTeXFuncConst b fc ++ "(" ++
                                    makeTeXElemTerm b et' ++ ")"
      VectToElem _ -> makeTeXElemTerm b $ expandDefinedElemTerm et

makeTeXElemTerms :: Bool -> [ElemTerm] -> String
makeTeXElemTerms b ets = foldr1 (\c d -> c ++ ",\\;" ++ d)
                                    (map (makeTeXElemTerm b) ets)
isAdditionV vt =
   case vt of
      vt1 :||||: vt2 -> True
      vt1 :&&&&: vt2 -> True
      _              -> False


makeTeXVectTerm :: Bool -> VectTerm -> String
makeTeXVectTerm b vt =
   case vt of
      VC vc          -> makeTeXVectConst b vc
      VV vv          -> makeTeXVectVari  b vv
      rt  :****: vt1 -> (if isAdditionR rt then "(" ++ (makeTeXRelaTerm b rt)
                                                ++ ")"
                                           else  makeTeXRelaTerm b rt        )  ++
                        "\\RELcompOP " ++
                        (if isAdditionV vt1 then "(" ++ (makeTeXVectTerm b vt1)
                                                 ++ ")"
                                            else makeTeXVectTerm b vt1        )
      vt1 :||||: vt2 -> makeTeXVectTerm b vt1 ++ "\\RELorOP "   ++
                                                 (makeTeXVectTerm b vt2)
      vt1 :&&&&: vt2 -> makeTeXVectTerm b vt1 ++ "\\RELandOP "  ++
                                                 (makeTeXVectTerm b vt2)
      Syq     rt vt1 -> "\\syq(" ++ makeTeXRelaTerm b rt ++ "," ++
                                    (makeTeXVectTerm b vt1) ++ ")"
      NegaV      vt1 -> "\\RELneg{" ++ (makeTeXVectTerm b vt1) ++ "}"
      NullV      o   -> case b of
                           True  -> "\\RELbot_{" ++ (makeTeXCatObject b o) ++ "}"
                           False -> "\\RELbot "
      UnivV      o   -> case b of
                           True  -> "\\RELtop_{" ++ (makeTeXCatObject b o) ++ "}"
                           False -> "\\RELtop "
      SupVect    vs  -> case vs of
```

```
                          --VarVS s o ->
                          VS vv fs -> let VarV s _ = vv
                                      in  "\\sup\\{ " ++ s ++ " \\mid " ++
                                          (makeTeXFormulae b fs) ++ "\\}"
                          VX vts _ -> "\\sup\\{ " ++ (makeTeXVectTerms b vts) ++
                                      "\\}"
      InfVect    vs  -> case vs of
                          --VarVS s o ->
                          VS vv fs  -> let VarV s _ = vv
                                       in  "\\inf\\{ " ++ s ++ " \\mid " ++
                                           (makeTeXFormulae b fs) ++ "\\}"
                          VX vts _ -> "\\inf\\{ " ++ (makeTeXVectTerms b vts) ++
                                      "\\}"
      RelaToVect _   -> makeTeXVectTerm b $ expandDefinedVectTerm vt
      PointVect  et  -> "(" ++ (makeTeXElemTerm b et) ++ "\\RELcompOP\\RELtop)"
      PowElemToVect _  -> makeTeXVectTerm b $ expandDefinedVectTerm vt
      VFctAppl vFct vt -> let VFCT vv vt1 = vFct
                              substArgsInFctl = substituteInVectTermAccordingToList
                                                ([],[],[(vv,vt)],[],[]) vt1
                          in  makeTeXVectTerm b substArgsInFctl


makeTeXVectTerms :: Bool -> [VectTerm] -> String
makeTeXVectTerms b vts = foldr1 (\c d -> c ++ ",\\;" ++ d)
                                (map (makeTeXVectTerm b) vts)

isAdditionR rt =
   case rt of
      rt1 :|||: rt2 -> True
      rt1 :&&&: rt2 -> True
      _             -> False

makeTeXRelaTerm :: Bool -> RelaTerm -> String   -- Bool distiguishes short/long text
makeTeXRelaTerm b rt =
   case rt of
      RC rc          -> makeTeXRelaConst b rc
      RV rv          -> makeTeXRelaVari  b rv
      rt1 :***: rt2  -> (if isAdditionR rt1 then "(" ++ (makeTeXRelaTerm b rt1)
                                                 ++ ")"
                                            else makeTeXRelaTerm b rt1        ) ++
                                                 "\\RELcompOP " ++
                        (if isAdditionR rt2 then "(" ++ (makeTeXRelaTerm b rt2)
                                                 ++ ")"
                                            else makeTeXRelaTerm b rt2        )
      rt1 :|||: rt2  -> makeTeXRelaTerm b rt1 ++ "\\RELorOP "   ++
                                                 (makeTeXRelaTerm b rt2)
      rt1 :&&&: rt2  -> makeTeXRelaTerm b rt1 ++ "\\RELandOP "  ++
                                                 (makeTeXRelaTerm b rt2)
      NegaR     rt1  -> "\\RELneg{" ++ (makeTeXRelaTerm b rt1) ++ "}"
      NullR     o o' -> case b of
```

```
                    True  -> "\\RELbot_{" ++ makeTeXCatObject b o  ++
                                             makeTeXCatObject b o' ++ "}"
                    False -> "\\RELbot "
UnivR     o o' -> case b of
                    True  -> "\\RELtop_{" ++ makeTeXCatObject b o  ++
                                             makeTeXCatObject b o' ++ "}"
                    False -> "\\RELtop "
Ident     o    -> case b of
                    True  -> "\\RELide_{" ++ makeTeXCatObject b o ++ "}"
                    False -> "\\RELide "
Convs  (rt1 :|||: rt2)  -> "(" ++ (makeTeXRelaTerm b (rt1 :|||: rt2)) ++
                                  ")\\RELtraOP "
Convs  (rt1 :&&&: rt2)  -> "(" ++ (makeTeXRelaTerm b (rt1 :&&&: rt2)) ++
                                  ")\\RELtraOP "
Convs  (rt1 :***: rt2)  -> "(" ++ (makeTeXRelaTerm b (rt1 :***: rt2)) ++
                                  ")\\RELtraOP "
Convs  (Convs  rt1) -> makeTeXRelaTerm b rt1
Convs     rt1  -> "{" ++ makeTeXRelaTerm b rt1 ++ "}\\RELtraOP "
vt1 :||--: vt2 -> makeTeXVectTerm b vt1 ++ "\\RELcompOP " ++
                  (makeTeXVectTerm b vt2) ++ "\\RELtraOP "
SupRela     rs ->
   case rs of
      VarRS s o o' -> let oText  = makeTeXCatObject b o
                          oText' = makeTeXCatObject b o'
                          lText  = "\\sup " ++ s ++ "_{(" ++
                                   oText ++ "," ++ oText' ++ ")}"
                          sText  = "\\sup " ++ s
                      in  if b then lText else sText
      RS rv fs -> let VarR s _ _ = rv
                  in  "\\sup\\{ " ++ s ++ " \\mid " ++
                      (makeTeXFormulae b fs) ++ "\\}"
      RX rts o1 o2 ->
         case null rts of
            True  -> case b of
                        True  -> "\\sup_{" ++
                                 makeTeXCatObject b o1 ++ ", " ++
                                 makeTeXCatObject b o2 ++ "}  \\emptyset "
                        False -> "\\sup \\emptyset "
            False -> "\\sup \\{" ++ (foldr1 (\x y -> x ++ ", " ++ y)
                     (map (makeTeXRelaTerm b) rts)) ++ "\\}"
InfRela     rs ->
   case rs of
      VarRS s o o' -> let oText  = makeTeXCatObject b o
                          oText' = makeTeXCatObject b o'
                          lText  = "\\inf " ++ s ++ "_{(" ++
                                   oText ++ "," ++ oText' ++ ")}"
                          sText  = "\\inf " ++ s
                      in  if b then lText else sText
      RS rv fs -> let varName rv =
                        case rv of
```

```
                              VarR s _ _ -> s
                              IndexedVarR s i _ _ -> s ++ "_{" ++ show i ++ "}"
                    in  "\\inf\\{ " ++ (varName rv) ++ " \\mid " ++
                        (makeTeXFormulae b fs) ++ "\\}"
      RX rts o1 o2 ->
         case null rts of
            True  -> case b of
                        True  -> "\\sup_{" ++
                                 makeTeXCatObject b o1 ++ ", " ++
                                 makeTeXCatObject b o2 ++ "}  \\emptyset "
                        False -> "\\sup \\emptyset "
            False -> "\\inf \\{" ++ (foldr1 (\x y -> x ++ ", " ++ y)
                     (map (makeTeXRelaTerm b) rts)) ++ "\\}"
Pi       o o'  -> case b of
                     True  -> "\\pi_{" ++ o1 ++ "\\times " ++
                              (makeTeXCatObject b o') ++
                              "," ++ o1 ++ "}"
                                 where o1 = makeTeXCatObject b o
                     False -> "\\pi"
Rho      o o'  -> case b of
                     True  -> "\\rho_{" ++ (makeTeXCatObject b o) ++
                              "\\times " ++ o2 ++ "," ++ o2 ++ "}"
                                 where o2 = makeTeXCatObject b o'
                     False -> "\\rho"
(:*:)  _ _     -> makeTeXRelaTerm b (expandDefinedRelaTerm rt)
(:\/:) _ _     -> makeTeXRelaTerm b (expandDefinedRelaTerm rt)
SyQ    rt1 rt2 -> "\\syq(" ++ makeTeXRelaTerm b rt1 ++ "," ++
                            (makeTeXRelaTerm b rt2) ++ ")"
Iota     o o'  -> case b of
                     True  -> "\\iota_{" ++ o1 ++ "," ++ o1 ++ "+" ++
                              (makeTeXCatObject b o') ++ "}"
                                 where o1 = makeTeXCatObject b o
                     False -> "\\iota"
Kappa    o o'  -> case b of
                     True  -> "\\kappa_{" ++ o2 ++ "," ++
                              (makeTeXCatObject b o) ++ "+" ++ o2 ++"}"
                                 where o2 = makeTeXCatObject b o'
                     False -> "\\kappa"
CASE   rt1 rt2 -> "if LEFT then " ++ (makeTeXRelaTerm b rt1)
                     ++ " else " ++ (makeTeXRelaTerm b rt2) ++ " fi "
Epsi     o     -> case b of
                     True  -> "\\epsilon_{" ++ (makeTeXCatObject b o) ++ "}"
                     False -> "\\epsilon "
PointDiag et   -> "(\\RELide\\RELandOP " ++ (makeTeXElemTerm b et) ++
                  "\\RELcompOP\\RELtop)"
InjTerm   vt   -> "(inject " ++ (makeTeXVectTerm b vt) ++ ")"
--Wait     rt1   ->
Belly pt         -> "As normal relation(" ++ makeTeXPartTerm b pt ++ ")"
ProdVectToRela _ -> makeTeXRelaTerm b $ expandDefinedRelaTerm rt
PartOrd po       -> "PartObj" ++ makeTeXParObject b po
```

```
      RFctAppl (RFCT argv rterm) argTerm ->
         case argv of
            EVar ea -> case argTerm of
                         ArgE ae -> makeTeXRelaTerm b $
                                       substituteInRelaTermAccordingToList
                                              ([],[(ea,ae)],[],[],[]) rterm
                         _       -> error "wrong argument type"
            VVar va -> case argTerm of
                         ArgV av -> makeTeXRelaTerm b $
                                       substituteInRelaTermAccordingToList
                                              ([],[],[(va,av)],[],[]) rterm
                         _       -> error "wrong argument type"
            RVar ra -> case argTerm of
                         ArgR ar -> makeTeXRelaTerm b $
                                       substituteInRelaTermAccordingToList
                                              ([],[],[],[(ra,ar)],[]) rterm
                         _       -> error "wrong argument type"



makeTeXRelaTerms :: Bool -> [RelaTerm] -> String
makeTeXRelaTerms b rts = foldr1 (\c d -> c ++ ",\\;" ++ d)
                                (map (makeTeXRelaTerm b) rts)

makeTeXPartTerm :: Bool -> PartTerm -> String
makeTeXPartTerm b rt =
   case rt of
      Lift   rt1  -> "Lift {" ++ makeTeXRelaTerm b rt1 ++ "}"
      Fetus po1 rt1 po2 ->
         let rtText  = makeTeXRelaTerm  b rt1
             po1Text = makeTeXParObject b po1
             po2Text = makeTeXParObject b po2
         in  "Fetus enclosed in Baby-Orderings(" ++ po1Text ++ "," ++  rtText ++
             "," ++ po2Text ++ ")"
      PPi    o o' -> "\\gPI_{(" ++ o1 ++ "\\times " ++ makeTeXParObject b o' ++
                        "," ++ o1 ++ ")}"   where o1 = makeTeXParObject b o
      PRho   o o' -> "\\gRho_{(" ++ o1 ++ "\\times " ++ makeTeXParObject b o' ++
                        "," ++ o1 ++ ")}"   where o1 = makeTeXParObject b o
      --PIota  o o' -> "\\gRho_{(" ++ o1 ++ "\\times " ++ makeTeXParObject b o' ++
      --                  "," ++ o1 ++ ")}"   where o1 = makeTeXParObject b o
      --PKappa o o' -> "\\gRho_{(" ++ o1 ++ "\\times " ++ makeTeXParObject b o' ++
      --                  "," ++ o1 ++ ")}"   where o1 = makeTeXParObject b o
      PEpsi   o    -> "\\EPS_{(" ++ o1 ++ ")}" where o1 = makeTeXParObject b o

makeTeXElemFct :: Bool -> ElemFct -> String
makeTeXElemFct b efct =
   case efct of
      EFCT ev et -> "\\langle\\backslash " ++ tEX b ev ++
                    "\\;\\longrightarrow\\;" ++ tEX b et ++ "\\rangle"
```

```
makeTeXVectFct :: Bool -> VectFct -> String
makeTeXVectFct b vfct =
   case vfct of
      VFCT vv vt -> "\\langle\\backslash " ++ tEX b vv ++
                    "\\;\\longrightarrow\\;" ++ tEX b vt ++ "\\rangle"
makeTeXRelaFct :: Bool -> RelaFct -> String
makeTeXRelaFct b rfct =
   case rfct of
      RFCT rv rt -> "\\langle\\backslash " ++ tEX b rv ++
                    "\\;\\longrightarrow\\;" ++ tEX b rt ++ "\\rangle"



makeTeXElemSET :: Bool -> ElemSET -> String
makeTeXElemSET b es =
   case es of
      VarES s _ -> s
      ET     o  -> makeTeXCatObject b o
      ES ev  fs -> "\\{ " ++ (makeTeXElemVari b ev) ++ " \\mid " ++
                   (makeTeXFormulae b fs) ++ "\\}"
      EX ets o  -> "\\{ " ++ (makeTeXElemTerms b ets) ++ "\\}"


makeTeXVectSET :: Bool -> VectSET -> String
makeTeXVectSET b vs =
   case vs of
      VarVS s _ -> s
      VS vv  fs -> "\\{ " ++ (makeTeXVectVari b vv) ++ " \\mid " ++
                   (makeTeXFormulae b fs) ++ "\\}"
      VX vts o  -> "\\{ " ++ (makeTeXVectTerms b vts) ++ "\\}"

makeTeXRelaSET ::  Bool -> RelaSET -> String
makeTeXRelaSET b rs =
   case rs of
      VarRS s o o' -> s
      RS rv fs     -> "\\{ " ++ (makeTeXRelaVari b rv) ++ " \\mid " ++
                      (makeTeXFormulae b fs) ++ "\\}"
      RX rts o o'  -> case null rts of
                         True  -> case b of
                                     True  -> "\\emptyset_{" ++
                                              makeTeXCatObject b o  ++ ", " ++
                                              makeTeXCatObject b o' ++ "}"
                                     False -> "\\emptyset "
                         False -> "\\{ " ++ (makeTeXRelaTerms b rts) ++ "\\}"

makeTeXElemForm :: Bool -> ElemForm -> String
makeTeXElemForm b ef =
   case ef of
      Equation t1 t2    -> makeTeXElemTerm b t1 ++ "=" ++
                           (makeTeXElemTerm b t2)
```

```
      NegaEqua t1 t2    -> makeTeXElemTerm b t1 ++ "\\not=" ++
                                   (makeTeXElemTerm b t2)
      QuantElemForm q v fs ->
         case b of
            True  -> "\\langle\\" ++ (if q == Univ then "forall " else "exists ") ++
                     (makeTeXElemVari  b v) ++ "\\in " ++
                     (makeTeXCatObject b (domEV v)) ++
                     ":\\ " ++ (makeTeXFormulae b fs) ++
                     "\\rangle "
            False -> "\\langle\\" ++ (if q == Univ then "forall " else "exists ") ++
                     (makeTeXElemVari b v) ++
                     ":\\ " ++ (makeTeXFormulae b fs) ++
                     "\\rangle "
makeTeXElemForms :: Bool -> [ElemForm] -> String
makeTeXElemForms b efs =
   foldr1 (\x y -> x ++ "$" ++ "\n\n\\bigskip\n\n" ++
                                   "$" ++ y) (map (makeTeXElemForm b) efs)


makeTeXVectForm :: Bool -> VectForm -> String
makeTeXVectForm b vf =
    case vf of
      vt1 :<===: vt2  -> makeTeXVectTerm b vt1 ++ "\\RELenthOP "  ++
                                                  (makeTeXVectTerm b vt2)
      _   :>===: _    -> makeTeXFormula b $ expandDefinedVectForm vf
      _   :====: _    -> makeTeXFormula b $ expandDefinedVectForm vf
      _   :<=/=: _    -> makeTeXFormula b $ expandDefinedVectForm vf
      _   :>=/=: _    -> makeTeXFormula b $ expandDefinedVectForm vf
      _   :==/=: _    -> makeTeXFormula b $ expandDefinedVectForm vf
      VectInSet vt vs -> makeTeXVectTerm b vt  ++ "\\in " ++
                                                 (makeTeXVectSET b vs)
      VE      vt t    -> makeTeXElemTerm b t ++ "\\in " ++
                                                 (makeTeXVectTerm b vt)
      QuantVectForm q vv fs ->
         case b of
            True  -> "\\langle\\" ++ (if q == Univ then "forall " else "exists ") ++
                     (makeTeXVectVari b vv) ++
                     "\\subseteq " ++ (makeTeXCatObject b (domVV vv)) ++
                     ":\\ " ++ (makeTeXFormulae b fs) ++ "\\rangle "
            False -> "\\langle\\" ++ (if q == Univ then "forall " else "exists ") ++
                     (makeTeXVectVari b vv) ++
                     ":\\ " ++ (makeTeXFormulae b fs) ++ "\\rangle "
makeTeXVectForms :: Bool -> [VectForm] -> String
makeTeXVectForms b vfs =
   foldr1 (\x y -> x ++ "$" ++ "\n\n\\bigskip\n\n" ++
                                   "$" ++ y) (map (makeTeXVectForm b) vfs)


makeTeXRelaForm :: Bool -> RelaForm -> String
makeTeXRelaForm b rf =
    case rf of
```

```
      rt1 :<==: rt2 -> makeTeXRelaTerm b rt1 ++ "\\RELenthOP "  ++
                                              (makeTeXRelaTerm b rt2)
      _   :>==: _   -> makeTeXFormula b $ expandDefinedRelaForm rf
      _   :===: _   -> makeTeXFormula b $ expandDefinedRelaForm rf
      _   :<=/: _   -> makeTeXFormula b $ expandDefinedRelaForm rf
      _   :>=/: _   -> makeTeXFormula b $ expandDefinedRelaForm rf
      _   :=/=: _   -> makeTeXFormula b $ expandDefinedRelaForm rf
      RelaInSet rt rs -> makeTeXRelaTerm b rt  ++ "\\in " ++
                                              (makeTeXRelaSET b rs)
      REE rt t1 t2    -> "(" ++ (makeTeXElemTerm b t1) ++ ", " ++
                                 (makeTeXElemTerm b t2)
                              ++ ")" ++ "\\in " ++ (makeTeXRelaTerm b rt)
      QuantRelaForm q rv fs ->
         case b of
            True  -> "\\langle\\" ++ (if q == Univ then "forall " else "exists ") ++
                     (makeTeXRelaVari b rv) ++ "\\subseteq " ++
                     (makeTeXCatObject b (domRV rv)) ++ "\\times " ++
                     (makeTeXCatObject b (codRV rv)) ++
                     ":\\langle " ++ (makeTeXFormulae b fs) ++
                     "\\rangle "
            False -> "\\langle\\" ++ (if q == Univ then "forall " else "exists ") ++
                     (makeTeXRelaVari b rv) ++
                     ":\\langle " ++ (makeTeXFormulae b fs) ++
                     "\\rangle "


makeTeXRelaForms :: Bool -> [RelaForm] -> String
makeTeXRelaForms b rfs =
   foldr1 (\x y -> x ++ "$" ++ "\n\n\\bigskip\n\n" ++ "$" ++ y)
                                    (map (makeTeXRelaForm b) rfs)

makeTeXFormVari :: Bool -> FormVari -> String
makeTeXFormVari _ fv =
   case fv of
      VarF          s   -> s
      IndexedVarF s i -> s ++ "_{" ++ (show i) ++ "}"

makeTeXFormula :: Bool -> Formula -> String
makeTeXFormula b ef =
   case ef of
      FV fv               -> makeTeXFormVari b fv
      EF ef               -> makeTeXElemForm b ef
      VF vf               -> makeTeXVectForm b vf
      RF rf               -> makeTeXRelaForm b rf
      Verum               -> "{\\tt True}"
      Falsum              -> "{\\tt False}"
      Negated  f1         -> "{\\tt \\neg }\\;(" ++ (makeTeXFormula b f1) ++ ")"
      Implies  f1 f2 ->
         let texF1 = makeTeXFormula b f1
             texF2 = makeTeXFormula b f2
```

```
         in  if   texF1 == texF2
             then "{\\tt True}"
             else "(" ++ texF1 ++ ")\\;\\longrightarrow\\;(" ++ texF2 ++ ")"

      SemEqu f1 f2 ->
         let texF1 = makeTeXFormula b f1
             texF2 = makeTeXFormula b f2
         in  if   texF1 == texF2
             then "{\\tt True}"
             else "(" ++ texF1 ++ ")\\;\\longleftrightarrow\\;(" ++ texF2 ++ ")"
      Disjunct f1 f2 ->
         let texF1 = makeTeXFormula b f1
             texF2 = makeTeXFormula b f2
         in  if   texF1 == texF2
             then texF1
             else "(" ++ texF1 ++ ")\\;\\predor\\;(" ++ texF2 ++ ")"

      Conjunct f1 f2 ->
         let texF1 = makeTeXFormula b f1
             texF2 = makeTeXFormula b f2
         in  if   texF1 == texF2
             then texF1
             else "(" ++ texF1 ++ ")\\;\\predand\\;(" ++ texF2 ++ ")"

makeTeXFormulae :: Bool -> [Formula] -> String
makeTeXFormulae b [] = "\\quad"
makeTeXFormulae b fs = foldr1 (\c d -> c ++ "$,\n\n\\bigskip\n$" ++ d)
                                  (map (makeTeXFormula b) fs)
```

Again we define a type class in order to switch to a shorter and unqualified notation.

```
class TeX a where
   tEX :: Bool -> a -> String
   rename :: ([(CatObjVar,CatObjVar)],
              [(ElemVari ,ElemVari )],[(VectVari,VectVari)],
              [(RelaVari ,RelaVari )],[(FormVari,FormVari)]) -> a -> a

instance TeX CatObject where
   tEX = makeTeXCatObject
   rename = renameCatObject
instance TeX ElemConst where
   tEX = makeTeXElemConst
   rename = renameElemConst
instance TeX ElemVari  where
   tEX = makeTeXElemVari
instance TeX VectConst where
   tEX = makeTeXVectConst
   rename = renameVectConst
instance TeX VectVari  where
```

```
   tEX = makeTeXVectVari
   rename = renameVectVari
instance TeX RelaConst where
   tEX = makeTeXRelaConst
   rename = renameRelaConst
instance TeX RelaVari  where
   tEX = makeTeXRelaVari
   rename = renameRelaVari
instance TeX EVRVari  where
   tEX = makeTeXEVRVari
   rename = renameEVRVari
instance TeX ElemTerm  where
   tEX = makeTeXElemTerm
   rename = renameElemTerm
instance TeX VectTerm  where
   tEX = makeTeXVectTerm
   rename = renameVectTerm
instance TeX RelaTerm  where
   tEX = makeTeXRelaTerm
   rename = renameRelaTerm
instance TeX ElemFct   where
   tEX = makeTeXElemFct
   rename = renameElemFct
instance TeX VectFct   where
   tEX = makeTeXVectFct
   rename = renameVectFct
instance TeX RelaFct   where
   tEX = makeTeXRelaFct
   rename = renameRelaFct
instance TeX ElemSET   where
   tEX = makeTeXElemSET
   rename = renameElemSET
instance TeX VectSET   where
   tEX = makeTeXVectSET
   rename = renameVectSET
instance TeX RelaSET   where
   tEX = makeTeXRelaSET
   rename = renameRelaSET
instance TeX ElemForm  where
   tEX = makeTeXElemForm
   rename = renameElemForm
instance TeX VectForm  where
   tEX = makeTeXVectForm
   rename = renameVectForm
instance TeX RelaForm  where
   tEX = makeTeXRelaForm
   rename = renameRelaForm
instance TeX FormVari  where
   tEX = makeTeXFormVari
   rename = renameFormVari
```

```
instance TeX Formula   where
   tEX = makeTeXFormula
   rename = renameFormula
```

We also provide a useful TEX representation for vectors, matrices and partialities.

```
makeTeXVectHoriz :: Bool -> [Bool] -> String
makeTeXVectHoriz _ v =
   "(" ++ foldl1 (\ x y  ->  x ++ " " ++ y) (map boolToChar v) ++ ")"
makeTeXVectVerti :: Bool -> [Bool] -> String
makeTeXVectVerti _ v =
   druckTeXMatH $ transpMat [v]


makeTeXMatr :: Bool -> [[Bool]] -> String
makeTeXMatr _ m = druckTeXMatH m

matMatToMat :: [[[[Bool]]]] -> [[Bool]]
matMatToMat mm =
   let combineMatrixRow mr = foldr1 (zipWith (++)) mr
   in  concatMap combineMatrixRow mm

makeTeXPartMoM :: Bool -> BabyMat -> String
makeTeXPartMoM _ bm =
   let BMMatOfMat moM   = convertBMToMatOfMat bm
       BMGlobal rD cD _ = convertBMToGlobal   bm
       numberToBoolVect n = take (n-1) (repeat False) ++ [True]
       rowD = concatMap numberToBoolVect rD
       colD = concatMap numberToBoolVect cD
   in  druckTeXMatLinesH rowD colD (matMatToMat moM)

makeTeXPartGlob :: Bool -> BabyMat -> String
makeTeXPartGlob _ bm =
   let BMGlobal rD cD g = convertBMToGlobal bm
       numberToBoolVect n = take (2^n-1) (repeat False) ++ [True]
       rowD = concatMap numberToBoolVect rD
       colD = concatMap numberToBoolVect cD
   in  druckTeXMatLinesH rowD colD g
```

### 4.5.3 Transformation to ASCII Constant-Width Form

It is sometimes helpful to get rid of all the typing information which is necessary to deal with the really difficult transformation tasks. So we design functions to strip most of the typing information just for readability and presentation purposes.

```
cutInPieces cs = cutIntoPieces 75 cs
cutIntoPieces nn cs =
   case length cs > nn of
      True  -> let (a,b) = splitAt nn cs
```

```
               in  case head b /= '_' of
                      True  -> a ++ " }" ++ "\n\\hbox{\\tt " ++ cutIntoPieces nn b
                      False -> cutIntoPieces (nn+1) cs
      False -> cs

cutInPieces1 cs = cutIntoPieces1 75 cs
cutIntoPieces1 nn cs =
   case length cs > nn of
      True  -> let (a,b) = splitAt nn cs
               in  case head b /= '_' of
                      True  -> a ++ " }" ++ "\n\\hbox{\\tt " ++ cutIntoPieces1 nn b
                      False -> cutIntoPieces1 (nn+1) cs
      False -> cs

prefixSlashToSpecChars "" = ""
prefixSlashToSpecChars (':':(' ':(' ':t))) =
            ":\\ " ++ prefixSlashToSpecChars t
prefixSlashToSpecChars (':':(' ':t)) =
            ":\\ " ++ prefixSlashToSpecChars t
prefixSlashToSpecChars (h:t) =
   case h of
      '_' -> "\\_" ++ prefixSlashToSpecChars t
      '&' -> "\\&" ++ prefixSlashToSpecChars t
      _   -> [h]   ++ prefixSlashToSpecChars t



prefixSlashToSpecChars1 "" = ""
prefixSlashToSpecChars1 (':':(' ':(' ':t))) =
            ":\\ " ++ prefixSlashToSpecChars1 t
prefixSlashToSpecChars1 (':':(' ':t)) =
            ":\\ " ++ prefixSlashToSpecChars1 t
prefixSlashToSpecChars1 (h:t) =
   case h of
      '_' -> "\\_" ++ prefixSlashToSpecChars1 t
      '&' -> "&"   ++ prefixSlashToSpecChars1 t
      _   -> [h]   ++ prefixSlashToSpecChars1 t


stripCatObject :: CatObject -> String
stripCatObject co =
   case co of
      OC (CstO oc) -> prefixSlashToSpecChars oc
      OV (VarO ov) -> prefixSlashToSpecChars ov
      OV (IndexedVarO ov i) -> prefixSlashToSpecChars ov ++ show i
      DirPro o o' -> stripCatObject o ++ "x" ++ stripCatObject o'
      DirSum o o' -> stripCatObject o ++ "+" ++ stripCatObject o'
      DirPow o    -> "P(" ++ stripCatObject o ++ ")"
      UnitOb      -> "()"
      QuotMod  rt -> stripCatObject (domRT rt) ++ "/" ++ "(" ++
```

```
                    (stripRelaTerm  rt) ++ ")"
      InjFrom  vt -> "SubObj(" ++ (stripVectTerm  vt) ++ ")"
      Strict   po -> "Strict(" ++ (stripParObject po) ++ ")"

stripParObject :: ParObject -> String
stripParObject po =
   case po of
      ParObj co      -> stripCatObject co ++ " lifted "
      ParPro po1 po2 -> stripParObject po1 ++ "px" ++ stripParObject po2
      ParSum po1 po2 -> stripParObject po1 ++ "p+" ++ stripParObject po2
      ParPow po1     -> "P(" ++ stripParObject po1 ++ ")"

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

stripElemVari :: ElemVari -> String
stripElemVari  ev =
   case ev of
      VarE          s   _ -> s
      IndexedVarE s i _ -> s ++ "_" ++ show i

stripElemTerm :: ElemTerm -> String
stripElemTerm et =
   case et of
      EC    ec -> " " ++ (stripElemConst ec) ++ " "
      EV    ev -> " " ++ (stripElemVari  ev) ++ " "
      Pair  et1 et2 -> "(" ++ stripElemTerm et1 ++ "," ++ stripElemTerm et2 ++ ")"
      Inj1 et  o    -> "in1(" ++ stripElemTerm et ++ ")"
      Inj2 o   et   -> "in2(" ++ stripElemTerm et ++ ")"
      ThatV vt -> "({\\tt That x} :   x   <- " ++ (stripVectTerm vt) ++ ") "
      SomeV vt -> "({\\tt Some x} :   x   <- " ++ (stripVectTerm vt) ++ ") "
      ThatR rt -> "({\\tt That x} : (x,x) <- " ++ (stripRelaTerm rt) ++ ") "
      SomeR rt -> "({\\tt Some x} : (x,x) <- " ++ (stripRelaTerm rt) ++ ") "
      FuncAppl fc et' -> stripFuncConst fc ++ "(" ++ stripElemTerm et'
      VectToElem _ -> stripElemTerm $ expandDefinedElemTerm et


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

stripVectVari :: VectVari -> String
stripVectVari  vv =
   case vv of
      VarV          s   _ -> s
      IndexedVarV s i _ -> s ++ "_" ++ show i

stripVectTerm :: VectTerm -> String
stripVectTerm vt =
   case vt of
      VC vc -> stripVectConst vc
```

```
      VV vv -> stripVectVari  vv
      rt1 :****: vt2 -> stripRelaTerm rt1 ++ ":****:" ++ (stripVectTerm vt2)
      --vt1 :||||: vt2 ->
      --vt1 :&&&&: vt2 ->
      --NegaV    vt1   ->
      NullV co       -> "NULL"
      UnivV co       -> "UNIV"
      --SupVect vs     ->
      --IndVect vs     ->
      --RelaToVect _   ->
      --PointVect et   ->
      --PowElemToVect _  ->
      Syq rt1 vt2 -> "Syq(" ++ stripRelaTerm rt1 ++ "," ++ (
                              stripVectTerm vt2) ++ ")"


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

stripFuncConst :: FuncConst -> String
stripFuncConst (Func s o o') = s

stripRelaVari :: RelaVari -> String
stripRelaVari rv =
   case rv of
      VarR          s   _ _ -> s
      IndexedVarR s i _ _ -> s ++ show i


stripRelaTerm :: RelaTerm -> String
stripRelaTerm rt =
   case rt of
      RC         rc  -> stripRelaConst rc
      RV         rv  -> stripRelaVari  rv
      rt1 :***:  rt2 -> stripRelaTerm rt1 ++ ":***:" ++ stripRelaTerm rt2
      rt1 :|||:  rt2 -> stripRelaTerm rt1 ++ ":|||:" ++ stripRelaTerm rt2
      rt1 :&&&:  rt2 -> stripRelaTerm rt1 ++ ":\\&\\&\\&:" ++ stripRelaTerm rt2
      NegaR      rt1 -> "NEGA(" ++ stripRelaTerm rt1 ++ ")"
      Ident      _   -> "IDEN"
      NullR      _ _ -> "NULL"
      UnivR      _ _ -> "UNIV"
      Convs      rt1 -> "TRAN(" ++ stripRelaTerm rt1 ++ ")"
      vt :||--:  vt' -> stripVectTerm vt ++ ":***:" ++ "TRAN(" ++
                        stripVectTerm vt' ++ ")"
      SupRela    rs  -> "sup" ++ stripRelaSET rs
      InfRela    rs  -> "inf" ++ stripRelaSET rs
      Pi       o o'  -> "PI(" ++ o1 ++ " x " ++ stripCatObject o' ++
                                "," ++ o1 ++ ")"   where o1 = stripCatObject o
      Rho      o o'  -> "RHO(" ++ stripCatObject o ++ " x " ++
                                o2 ++ "," ++ o2 ++ ")"   where o2 = stripCatObject o'
      (:*:)    _ _   -> stripRelaTerm $ expandDefinedRelaTerm rt
```

```
      (:\/:)   _ _   -> stripRelaTerm $ expandDefinedRelaTerm rt
      SyQ rt1 rt2    ->  "SyQ(" ++ stripRelaTerm rt1 ++ "," ++
                                   (stripRelaTerm rt2) ++ ")"
      Iota     o o'  -> "IOTA(" ++ o1 ++ "," ++ o1 ++ " + " ++ stripCatObject o' ++
                            ")"   where o1 = stripCatObject o
      Kappa    o o'  -> "KAPPA(" ++ o2 ++ "," ++ stripCatObject o ++ " + " ++ o2 ++
                            ")"   where o2 = stripCatObject o'
      CASE   rt1 rt2 -> "if LEFT then " ++ (stripRelaTerm rt1)
                           ++ " else " ++ (stripRelaTerm rt2) ++ " fi "
      Epsi     o     -> "EPSI(" ++ stripCatObject o ++ ")"
      PointDiag  et  -> "PointDiag(" ++ stripElemTerm et ++ ")"
      InjTerm    vt  -> "InjTerm("   ++ stripVectTerm vt ++ ")"
      ProdVectToRela _ -> stripRelaTerm $ expandDefinedRelaTerm rt
      --Wait     rt1   -> "Wait(" ++ stripPartTerm rt1 ++ ")"

stripRelaTerms :: [RelaTerm] -> String
stripRelaTerms []    = ""
stripRelaTerms [rt]  = stripRelaTerm rt
stripRelaTerms (h:t) = stripRelaTerm h ++ ",\n " ++ stripRelaTerms t

stripPartTerm :: PartTerm -> String
stripPartTerm rt =
   case rt of
      Lift rt1 -> "Lift(" ++ stripRelaTerm rt1 ++ ")"
      Fetus po1 rt1 po2 ->
         let rtText  = stripRelaTerm rt1
             po1Text = stripParObject po1
             po2Text = stripParObject po2
         in  "Fetus enclosed in Baby-Orderings(" ++ po1Text ++
             "," ++  rtText ++ "," ++ po2Text ++ ")"
      PPi    o o' -> "\\gPI_{(" ++ o1 ++ "\\times " ++ stripParObject o' ++
                         "," ++ o1 ++ ")}"   where o1 = stripParObject o
      PRho   o o' -> "\\gRho_{(" ++ stripParObject o ++ "\\times " ++ o2 ++
                         "," ++ o2 ++ ")}"   where o2 = stripParObject o'
      --PIota  o o' -> "\\gRho_{(" ++ o1 ++ "\\times " ++ stripParObject o' ++
      --                   "," ++ o1 ++ ")}"   where o1 = stripParObject o
      --PKappa o o' -> "\\gRho_{(" ++ o1 ++ "\\times " ++ stripParObject o' ++
      --                   "," ++ o1 ++ ")}"   where o1 = stripParObject o
      PEpsi   o    -> "EPS_{(" ++ o1 ++ ")}" where o1 = stripParObject o


stripRelaSET :: RelaSET -> String
stripRelaSET rs =
   case rs of
      VarRS s o o'  -> s
      RS rv  fs     -> "{ " ++ stripRelaVari  rv  ++ " | " ++
                       stripFormulae fs ++ " }"
      RX rts o  o'  -> "{ " ++ stripRelaTerms rts ++ " }"

stripElemForm :: ElemForm -> String
```

```
stripElemForm ef =
   case ef of
      Equation et1 et2         -> (stripElemTerm et1) ++ " = " ++
                                  (stripElemTerm et2)
      NegaEqua et1 et2         -> (stripElemTerm et1) ++ "/= " ++
                                  (stripElemTerm et2)
      QuantElemForm q ev fs -> (if q == Univ then "FORALL "
                                             else "EXISTS ") ++ (stripElemVari ev) ++
                       ": " ++ (stripFormulae fs)

stripVectForm :: VectForm -> String
stripVectForm vf =
   case vf of
      vt1 :<===: vt2   -> stripVectTerm vt1 ++ " :<===: " ++ stripVectTerm vt2
      vt1 :>===: vt2   -> stripVectTerm vt1 ++ " :>===: " ++ stripVectTerm vt2
      vt1 :====: vt2   -> stripVectTerm vt1 ++ " :====: " ++ stripVectTerm vt2
      vt1 :<=/=: vt2   -> stripVectTerm vt1 ++ " :<=/=: " ++ stripVectTerm vt2
      vt1 :>=/=: vt2   -> stripVectTerm vt1 ++ " :>=/=: " ++ stripVectTerm vt2
      vt1 :==/=: vt2   -> stripVectTerm vt1 ++ " :==/=: " ++ stripVectTerm vt2
      VE     vt t -> (stripVectTerm vt) ++ "(" ++ (stripElemTerm t) ++ ")"
      QuantVectForm q vv fs -> (if q == Univ then "FORALL " else
                                             "EXISTS ") ++ (stripVectVari vv) ++
                                  ": " ++ (stripFormulae fs)


stripRelaForm :: RelaForm -> String
stripRelaForm rf =
    case rf of
      rt1 :<==: rt2   -> stripRelaTerm rt1 ++ " :<==: " ++ stripRelaTerm rt2
      rt1 :>==: rt2   -> stripRelaTerm rt1 ++ " :>==: " ++ stripRelaTerm rt2
      rt1 :===: rt2   -> stripRelaTerm rt1 ++ " :===: " ++ stripRelaTerm rt2
      rt1 :<=/: rt2   -> stripRelaTerm rt1 ++ " :<=/: " ++ stripRelaTerm rt2
      rt1 :>=/: rt2   -> stripRelaTerm rt1 ++ " :>=/: " ++ stripRelaTerm rt2
      rt1 :=/=: rt2   -> stripRelaTerm rt1 ++ " :=/=: " ++ stripRelaTerm rt2
      RelaInSet rt rs -> stripRelaTerm rt ++ " in " ++ stripRelaSET rs
      REE    rt t1 t2 -> (stripRelaTerm rt) ++ "(" ++ (stripElemTerm t1) ++
                         "," ++ (stripElemTerm t2) ++ ")"
      --UnivQuantRelaForm  rv rf -> (domRV rv, codRV rv)
      --ExistQuantRelaForm rv rf -> (domRV rv, codRV rv)

stripRelaForms :: [RelaForm] -> String
stripRelaForms rfs = concatMap stripRelaForm rfs

stripFormula :: Formula -> String
stripFormula f =
   case f of
      EF ef          -> stripElemForm ef
      VF vf          -> stripVectForm vf
      RF rf          -> stripRelaForm rf
      --PF pf          -> stripPartForm pf
      Verum          -> "True"
```

```
      Falsum         -> "False"
      Negated  f1    -> "NOT(" ++ stripFormula f1 ++ ")"
      Implies  f1 f2 -> stripFormula f1 ++ " ===> "  ++ stripFormula f2
      SemEqu   f1 f2 -> stripFormula f1 ++ " <===> " ++ stripFormula f2
      Disjunct f1 f2 -> stripFormula f1 ++ " OR "    ++ stripFormula f2
      Conjunct f1 f2 -> stripFormula f1 ++ " AND "   ++ stripFormula f2

stripFormulae :: [Formula] -> String
stripFormulae [] = "?  ?"
stripFormulae fs = foldr1 (\x y -> x ++ ", " ++ y) (map stripFormula fs)
```

## 4.6 Normal Form

The normal form is defined following the traditional line of getting a formula with prenex quantifications. Then a conjunction of disjunctions for the quantifier-free part must be reached. Finally, negations shall disappear eliminating two consecutive ones and by rolling negation over to the relational terms, which are, thus, considered basic constituents. The formal definition of the normal form may directly be obtained from the functions to test whether a formula belongs to the sublanguage generated by the grammar thus indicated.

```
isBasicFormula :: Formula -> Bool
isBasicFormula f =
   case f of
      -- OF
      EF ef -> isBasicElemForm ef
      VF vf -> isBasicVectForm vf
      RF rf -> isBasicRelaForm rf
      _     -> False

isBasicElemForm :: ElemForm -> Bool
isBasicElemForm ef =
   case ef of
      Equation _ _ -> True
      NegaEqua _ _ -> True
      _            -> False

isBasicVectForm :: VectForm -> Bool
isBasicVectForm vf =
   case vf of
      VE _ _     -> True
      _ :====: _ -> True
      _ :<===: _ -> True
      _ :>===: _ -> True
      _ :==/=: _ -> True
      _ :<=/=: _ -> True
      _ :>=/=: _ -> True
      _          -> False
```

```
isBasicRelaForm :: RelaForm -> Bool
isBasicRelaForm rf =
   case rf of
      REE _ _ _ -> True
      _ :===: _ -> True
      _ :<==: _ -> True
      _ :>==: _ -> True
      _ :=/=: _ -> True
      _ :<=/: _ -> True
      _ :>=/: _ -> True
      _         -> False

isDisjOfBasicForms :: Formula -> Bool
isDisjOfBasicForms f =
   case f of
      Disjunct f1 f2 -> isDisjOfBasicForms f1 && isDisjOfBasicForms f2
      _              -> isBasicFormula f

isConjOfDisjOfBasicForms :: Formula -> Bool
isConjOfDisjOfBasicForms f =
   case f of
      Conjunct f1 f2 -> isConjOfDisjOfBasicForms f1 && isConjOfDisjOfBasicForms f2
      _              -> isDisjOfBasicForms f

isMixQuOfConjOfDisjOfBasicForms :: Formula -> Bool
isMixQuOfConjOfDisjOfBasicForms f =
   case f of
      EF (QuantElemForm _ _ fs) ->
         if   null fs then False
         else if not $ null $ tail fs then False
         else isMixQuOfConjOfDisjOfBasicForms (head fs)
      VF (QuantVectForm _ _ fs) ->
         if   null fs then False
         else if not $ null $ tail fs then False
         else isMixQuOfConjOfDisjOfBasicForms (head fs)
      RF (QuantRelaForm _ _ fs) ->
         if   null fs then False
         else if not $ null $ tail fs then False
         else isMixQuOfConjOfDisjOfBasicForms (head fs)
      _ -> isConjOfDisjOfBasicForms f
```

## 4.7 Transformation to Normal Form

For several reasons it is a good idea to offer a program that transforms a formula into the normal form just defined.

```
formulaToNormalForm :: Formula -> Formula
formulaToNormalForm f =
```

```
   case f of
      -- OF
      EF ef  -> EF $ elemFormulaToNormalForm ef
      VF vf  -> VF $ vectFormulaToNormalForm vf
      RF rf  -> RF $ relaFormulaToNormalForm rf
      PF _   -> error "NormalForm for partiality not programmed"
      Verum  -> Verum
      Falsum -> Falsum
      Negated  f1    -> pushDownNegationInFormula
                        (formulaToNormalForm f1)
      Implies  f1 f2 -> pushDownDisjunctionInFormula
                        (pushDownNegationInFormula $
                         formulaToNormalForm f1)
                        (formulaToNormalForm f2 )
      SemEqu   f1 f2 -> formulaToNormalForm $
                        Conjunct (Implies f1 f2) (Implies f2 f1)
      Disjunct f1 f2 -> pushDownDisjunctionInFormula
                        (formulaToNormalForm f1)
                        (formulaToNormalForm f2)
      Conjunct f1 f2 -> pushDownConjunctionInFormula
                        (formulaToNormalForm f1)
                        (formulaToNormalForm f2)

elemFormulaToNormalForm :: ElemForm -> ElemForm
elemFormulaToNormalForm ef =
   case ef of
      Equation _ _  -> ef
      NegaEqua _ _  -> ef
      QuantElemForm q ev fs ->
         QuantElemForm q ev [formulaToNormalForm (foldr1 Conjunct fs)]

vectFormulaToNormalForm :: VectForm -> VectForm
vectFormulaToNormalForm vf =
   case vf of
      QuantVectForm q vv fs ->
         QuantVectForm q vv [formulaToNormalForm (foldr1 Conjunct fs)]
      -- VectInSET
      _ -> vf

relaFormulaToNormalForm :: RelaForm -> RelaForm
relaFormulaToNormalForm rf =
   case rf of
      QuantRelaForm q rv fs ->
         QuantRelaForm q rv [formulaToNormalForm (foldr1 Conjunct fs)]
      -- RelaInSET
      _ -> rf
```

As one could see, there are tasks to push negation, disjunction, and conjunction down only when the arguments are supposed to already be in normal form. This requires some case analyses.

```
pushDownNegationInFormula :: Formula -> Formula
```

```
        -- argument in NF by construction
pushDownNegationInFormula f =
   case f of
      EF ef -> EF $ pushDownNegationElemForm ef
      VF vf -> VF $ pushDownNegationVectForm vf
      RF rf -> RF $ pushDownNegationRelaForm rf
      Disjunct f1 f2 -> pushDownConjunctionInFormula
                           (pushDownNegationInFormula f1)
                           (pushDownNegationInFormula f2)
      Conjunct f1 f2 -> pushDownDisjunctionInFormula
                           (pushDownNegationInFormula f1)
                           (pushDownNegationInFormula f2)

pushDownNegationElemForm :: ElemForm -> ElemForm
        -- argument in NF by construction
pushDownNegationElemForm ef =
   case ef of
      Equation et1 et2 -> NegaEqua et1 et2
      NegaEqua et1 et2 -> Equation et1 et2
      QuantElemForm q ev [f'] ->
         QuantElemForm (switchUE q) ev [pushDownNegationInFormula f']

pushDownNegationVectForm :: VectForm -> VectForm
        -- argument in NF by construction
pushDownNegationVectForm vf =
   case vf of
      vt1 :<===: vt2 -> vt1 :<=/=: vt2
      vt1 :>===: vt2 -> vt1 :>=/=: vt2
      vt1 :====: vt2 -> vt1 :==/=: vt2
      vt1 :==/=: vt2 -> vt1 :====: vt2
      vt1 :<=/=: vt2 -> vt1 :<===: vt2
      vt1 :>=/=: vt2 -> vt1 :>===: vt2
      -- VectInSet ->
      VE vt et -> VE (NegaV vt) et
      QuantVectForm q vv [f'] ->
         QuantVectForm (switchUE q) vv [pushDownNegationInFormula f']

pushDownNegationRelaForm :: RelaForm -> RelaForm
        -- argument in NF by construction
pushDownNegationRelaForm rf =
   case rf of
      rt1 :<==: rt2  -> rt1 :<=/: rt2
      rt1 :>==: rt2  -> rt1 :>=/: rt2
      rt1 :===: rt2  -> rt1 :=/=: rt2
      rt1 :<=/: rt2  -> rt1 :<==: rt2
      rt1 :>=/: rt2  -> rt1 :>==: rt2
      rt1 :=/=: rt2  -> rt1 :===: rt2
      -- RelaInSet ->
      REE rt et1 et2 -> REE (NegaR rt) et1 et2
      QuantRelaForm q rv [f'] ->
```

```
         QuantRelaForm (switchUE q) rv [pushDownNegationInFormula f']

rebuildFormWithNewElemVari :: [ElemVari] ->
                               ElemVari  -> Formula -> (ElemVari,Formula)
rebuildFormWithNewElemVari evs ev f =
   let eNew = nextElemVari (dom ev) evs
       fNew = substituteInFormulaAccordingToList ([],[(ev,EV eNew)],[],[],[]) f
   in  (eNew,fNew)

rebuildFormWithNewVectVari :: [VectVari] ->
                               VectVari  -> Formula -> (VectVari,Formula)
rebuildFormWithNewVectVari vvs vv f =
   let vNew = nextVectVari (dom vv) vvs
       fNew = substituteInFormulaAccordingToList ([],[],[(vv,VV vNew)],[],[]) f
   in  (vNew,fNew)

rebuildFormWithNewRelaVari :: [RelaVari] ->
                               RelaVari  -> Formula -> (RelaVari,Formula)
rebuildFormWithNewRelaVari rvs rv f =
   let rNew = nextRelaVari (dom rv) (cod rv) rvs
       fNew = substituteInFormulaAccordingToList ([],[],[],[(rv,RV rNew)],[]) f
   in  (rNew,fNew)

pushDownDisjunctionInFormula :: Formula -> Formula -> Formula
        -- both arguments in NF by construction
pushDownDisjunctionInFormula f1 f2 =
   let (_,_,evs,_,vvs,_,rvs,_,_,_) = syntMatUsedInFormulae [f1,f2]
       isBas1 = isBasicFormula f1
       isBas2 = isBasicFormula f2
       isDis1 = isDisjOfBasicForms f1
       isDis2 = isDisjOfBasicForms f2
       isCon1 = isConjOfDisjOfBasicForms f1
       isCon2 = isConjOfDisjOfBasicForms f2
       isQua1 = isMixQuOfConjOfDisjOfBasicForms f1
       isQua2 = isMixQuOfConjOfDisjOfBasicForms f2
       switch = (isBas2 && not isBas1) ||
                (isDis2 && not isBas2 && not isDis1) ||
                (isCon2 && not isDis2 && not isCon1)
   in  if      f1 == Verum || f2 == Verum then Verum
       else if f1 == Falsum then f2
       else if f2 == Falsum then f1
       else case switch of
              True  -> pushDownDisjunctionInFormula f2 f1
              False -> if isDis1 && isDis2 then Disjunct f1 f2
                  else if isDis1 && isCon2
                        then let Conjunct f21 f22 = f2
                             in  Conjunct (Disjunct f1 f21) (Disjunct f1 f22)
                  else if isCon1 && not isDis1 && isCon2 && not isDis2
                        then let Conjunct f11 f12 = f1
                                 Conjunct f21 f22 = f2
```

```
            in  Conjunct
                 (Conjunct (Disjunct f11 f21) (Disjunct f11 f22))
                 (Conjunct (Disjunct f12 f21) (Disjunct f12 f22))
else if isCon1 && not isCon2 then
      case f2 of
         EF (QuantElemForm q ev2 [f2']) ->
            let (eNew2,fNew2) = rebuildFormWithNewElemVari
                                evs ev2 f2'
            in  EF (QuantElemForm q eNew2
                    [pushDownDisjunctionInFormula f1 fNew2])
         VF (QuantVectForm q vv2 [f2']) ->
            let (vNew2,fNew2) = rebuildFormWithNewVectVari
                                vvs vv2 f2'
            in  VF (QuantVectForm q vNew2
                    [pushDownDisjunctionInFormula f1 fNew2])
         RF (QuantRelaForm q rv2 [f2']) ->
            let (rNew2,fNew2) = rebuildFormWithNewRelaVari
                                rvs rv2 f2'
            in  RF (QuantRelaForm q rNew2
                    [pushDownDisjunctionInFormula f1 fNew2])
else if not isCon1 && not isCon2 then
      case f1 of
         EF (QuantElemForm q ev1 [f1']) ->
            let (eNew1,fNew1) = rebuildFormWithNewElemVari
                                evs ev1 f1'
            in  case f2 of
                 EF (QuantElemForm q2 ev2 [f2']) ->
                  let (eNew2,fNew2) = rebuildFormWithNewElemVari
                                      (eNew1:evs) ev2 f2'
                      fNew3 = EF (QuantElemForm q2 eNew2
                                 [pushDownDisjunctionInFormula
                                  fNew1 fNew2])
                  in  EF (QuantElemForm q eNew1 [fNew3])
                 VF (QuantVectForm q2 vv2 [f2']) ->
                  let (vNew2,fNew2) = rebuildFormWithNewVectVari
                                      vvs vv2 f2'
                      fNew3 = VF (QuantVectForm q2 vNew2
                                 [pushDownDisjunctionInFormula
                                  fNew1 fNew2])
                  in  EF (QuantElemForm q eNew1 [fNew3])
                 RF (QuantRelaForm q2 rv2 [f2']) ->
                  let (rNew2,fNew2) = rebuildFormWithNewRelaVari
                                      rvs rv2 f2'
                      fNew3 = RF (QuantRelaForm q2 rNew2
                                 [pushDownDisjunctionInFormula
                                  fNew1 fNew2])
                  in  EF (QuantElemForm q eNew1 [fNew3])
         VF (QuantVectForm q vv1 [f1']) ->
            let (vNew1,fNew1) = rebuildFormWithNewVectVari
                                vvs vv1 f1'
```

```
                    in  case f2 of
                         EF (QuantElemForm q2 ev2 [f2']) ->
                          let (eNew2,fNew2) = rebuildFormWithNewElemVari
                                              evs ev2 f2'
                              fNew3 = EF (QuantElemForm q2 eNew2
                                        [pushDownDisjunctionInFormula
                                       fNew1 fNew2])
                          in  VF (QuantVectForm q vNew1 [fNew3])
                         VF (QuantVectForm q2 vv2 [f2']) ->
                          let (vNew2,fNew2) = rebuildFormWithNewVectVari
                                              (vNew1:vvs) vv2 f2'
                              fNew3 = VF (QuantVectForm q2 vNew2
                                        [pushDownDisjunctionInFormula
                                       fNew1 fNew2])
                          in  VF (QuantVectForm q vNew1 [fNew3])
                         RF (QuantRelaForm q2 rv2 [f2']) ->
                          let (rNew2,fNew2) = rebuildFormWithNewRelaVari
                                              rvs rv2 f2'
                              fNew3 = RF (QuantRelaForm q2 rNew2
                                        [pushDownDisjunctionInFormula
                                       fNew1 fNew2])
                          in  VF (QuantVectForm q vNew1 [fNew3])
                RF (QuantRelaForm q rv1 [f1']) ->
                    let (rNew1,fNew1) = rebuildFormWithNewRelaVari
                                        rvs rv1 f1'
                    in  case f2 of
                         EF (QuantElemForm q2 ev2 [f2']) ->
                          let (eNew2,fNew2) = rebuildFormWithNewElemVari
                                              evs ev2 f2'
                              fNew3 = EF (QuantElemForm q2 eNew2
                                        [pushDownDisjunctionInFormula
                                       fNew1 fNew2])
                          in  RF (QuantRelaForm q rNew1 [fNew3])
                         VF (QuantVectForm q2 vv2 [f2']) ->
                          let (vNew2,fNew2) = rebuildFormWithNewVectVari
                                              vvs vv2 f2'
                              fNew3 = VF (QuantVectForm q2 vNew2
                                        [pushDownDisjunctionInFormula
                                       fNew1 fNew2])
                          in  RF (QuantRelaForm q rNew1 [fNew3])
                         RF (QuantRelaForm q2 rv2 [f2']) ->
                          let (rNew2,fNew2) = rebuildFormWithNewRelaVari
                                              (rNew1:rvs) rv2 f2'
                              fNew3 = RF (QuantRelaForm q2 rNew2
                                        [pushDownDisjunctionInFormula
                                       fNew1 fNew2])
                          in  RF (QuantRelaForm q rNew1 [fNew3])
          else Verum -- should not be reached!
                     -- Only to make the guard visible after else
```

```
pushDownConjunctionInFormula :: Formula -> Formula -> Formula
        -- both arguments in NF by construction
pushDownConjunctionInFormula f1 f2 =
   let (_,_,evs,_,vvs,_,rvs,_,_,_) = syntMatUsedInFormulae [f1,f2]
       isBas1 = isBasicFormula f1
       isBas2 = isBasicFormula f2
       isDis1 = isDisjOfBasicForms f1
       isDis2 = isDisjOfBasicForms f2
       isCon1 = isConjOfDisjOfBasicForms f1
       isCon2 = isConjOfDisjOfBasicForms f2
       isQua1 = isMixQuOfConjOfDisjOfBasicForms f1
       isQua2 = isMixQuOfConjOfDisjOfBasicForms f2
       switch = (isBas2 && not isBas1) ||
                (isDis2 && not isDis1) ||
                (isCon2 && not isCon1)
   in  if      f1 == Falsum || f2 == Falsum then Falsum
       else if f1 == Verum then f2
       else if f2 == Verum then f1
       else case switch of
          True  -> pushDownConjunctionInFormula f2 f1
          False -> if      isCon1 && isCon2 then Conjunct f1 f2
                   else if not isCon1 && not isCon2 then
                         case f1 of
                            EF (QuantElemForm q ev1 [f1']) ->
                               let (eNew1,fNew1) = rebuildFormWithNewElemVari
                                                   evs ev1 f1'
                               in  case f2 of
                                    EF (QuantElemForm q2 ev2 [f2']) ->
                                     let (eNew2,fNew2) = rebuildFormWithNewElemVari
                                                         (eNew1:evs) ev2 f2'
                                         fNew3 = EF (QuantElemForm q2 eNew2
                                                   [pushDownConjunctionInFormula
                                                fNew1 fNew2])
                                     in  EF (QuantElemForm q eNew1 [fNew3])
                                    VF (QuantVectForm q2 vv2 [f2']) ->
                                     let (vNew2,fNew2) = rebuildFormWithNewVectVari
                                                         vvs vv2 f2'
                                         fNew3 = VF (QuantVectForm q2 vNew2
                                                   [pushDownConjunctionInFormula
                                                fNew1 fNew2])
                                     in  EF (QuantElemForm  q eNew1 [fNew3])
                                    RF (QuantRelaForm q2 rv2 [f2']) ->
                                     let (rNew2,fNew2) = rebuildFormWithNewRelaVari
                                                         rvs rv2 f2'
                                         fNew3 = RF (QuantRelaForm q2 rNew2
                                                   [pushDownConjunctionInFormula
                                                fNew1 fNew2])
                                     in  EF (QuantElemForm q eNew1 [fNew3])
                            VF (QuantVectForm q vv1 [f1']) ->
                               let (vNew1,fNew1) = rebuildFormWithNewVectVari
```

```
                                    vvs vv1 f1'
                    in  case f2 of
                         EF (QuantElemForm q2 ev2 [f2']) ->
                          let (eNew2,fNew2) = rebuildFormWithNewElemVari
                                              evs ev2 f2'
                              fNew3 = EF (QuantElemForm q2 eNew2
                                         [pushDownConjunctionInFormula
                                       fNew1 fNew2])
                          in  VF (QuantVectForm q vNew1 [fNew3])
                         VF (QuantVectForm q2 vv2 [f2']) ->
                          let (vNew2,fNew2) = rebuildFormWithNewVectVari
                                              (vNew1:vvs) vv2 f2'
                              fNew3 = VF (QuantVectForm q2 vNew2
                                         [pushDownConjunctionInFormula
                                       fNew1 fNew2])
                          in  VF (QuantVectForm q vNew1 [fNew3])
                         RF (QuantRelaForm q2 rv2 [f2']) ->
                          let (rNew2,fNew2) = rebuildFormWithNewRelaVari
                                              rvs rv2 f2'
                              fNew3 = RF (QuantRelaForm q2 rNew2
                                         [pushDownConjunctionInFormula
                                       fNew1 fNew2])
                          in  VF (QuantVectForm q vNew1 [fNew3])
                RF (QuantRelaForm q rv1 [f1']) ->
                    let (rNew1,fNew1) = rebuildFormWithNewRelaVari
                                        rvs rv1 f1'
                    in  case f2 of
                         EF (QuantElemForm q2 ev2 [f2']) ->
                          let (eNew2,fNew2) = rebuildFormWithNewElemVari
                                              evs ev2 f2'
                              fNew3 = EF (QuantElemForm q2 eNew2
                                         [pushDownConjunctionInFormula
                                       fNew1 fNew2])
                          in  RF (QuantRelaForm q rNew1 [fNew3])
                         VF (QuantVectForm q2 vv2 [f2']) ->
                          let (vNew2,fNew2) = rebuildFormWithNewVectVari
                                              vvs vv2 f2'
                              fNew3 = VF (QuantVectForm q2 vNew2
                                         [pushDownConjunctionInFormula
                                       fNew1 fNew2])
                          in  RF (QuantRelaForm q rNew1 [fNew3])
                         RF (QuantRelaForm q2 rv2 [f2']) ->
                          let (rNew2,fNew2) = rebuildFormWithNewRelaVari
                                              (rNew1:rvs) rv2 f2'
                              fNew3 = RF (QuantRelaForm q2 rNew2
                                         [pushDownConjunctionInFormula
                                       fNew1 fNew2])
                          in  RF (QuantRelaForm q rNew1 [fNew3])
else case f2 of
       EF (QuantElemForm q ev2 [f2']) ->
```

```
   let (eNew2,fNew2) = rebuildFormWithNewElemVari
                                evs ev2 f2'
   in  EF (QuantElemForm q eNew2
          [pushDownConjunctionInFormula f1 fNew2])
VF (QuantVectForm q vv2 [f2']) ->
   let (vNew2,fNew2) = rebuildFormWithNewVectVari
                                vvs vv2 f2'
   in  VF (QuantVectForm q vNew2
          [pushDownConjunctionInFormula f1 fNew2])
RF (QuantRelaForm q rv2 [f2']) ->
   let (rNew2,fNew2) = rebuildFormWithNewRelaVari
                                rvs rv2 f2'
   in  RF (QuantRelaForm q rNew2
       [pushDownConjunctionInFormula f1 fNew2])
```

# 5 Standard Examples

It is advisable to have numerous examples that may routinely be run testwise during the development of a system of this size. To this end we formulate the Dedekind and Schröder rules together with the well-known formulae characterizing direct sum, product, and power. Also their translations and well-formedness predicates are collected here.

## 5.1 Dedekind and Schröder Formulae

As an example we consider the Dedekind formula. It is first built without care on typing, i.e., at every point a new type is assumed. Then we correct these types according to the restrictions the architecture of the Dedekind construct imposes and get the correctly typed version. This is then used in other examples.

```
dedekindForm sI =
   let (_,_,_,rvs,_) = supply sI 0 0 0 3 0
       [p,q,r] = map RV rvs
       left    = p :***: q :&&&: r
       right1  = p :&&&: (r :***: (Convs q))
       right2  = q :&&&: (Convs p :***: r)
       dedeRaw = left :<==: (right1 :***: right2)
       dedeTyp = generalTypeOfFormula (RF dedeRaw)
       dede    = renameSingle dedeTyp
       firstOrderDede         = translateFormula (VU [] [] [] [] []) dede
       firstOrderDedeTeXLONG  = tEX True  firstOrderDede
       firstOrderDedeTeXSHORT = tEX False firstOrderDede
   in  (dede, isWellFormed dede, tEX False dede, tEX True dede,
        stripFormula dede, firstOrderDede,
        firstOrderDedeTeXSHORT, firstOrderDedeTeXLONG)
```

The following gives names to the details of the result, which will be used several times in other contexts.

```
(correctDedekindFormula,isCorrectDedekind,teXDedekindshort,teXDedekindlong,
 dedekindInASCII,firstOrderDedekind,
 firstOrderDedekindTeXSHORT,firstOrderDedekindTeXLONG) = dedekindForm 987
```

The latter lines produce results as follows:

```
isCorrectDedekind: True
```

teXDedekindshort: $P;Q \cap R \subseteq (P \cap R;Q^{\mathsf{T}});(Q \cap P^{\mathsf{T}};R)$

teXDedekindlong
$P_{O_1,O_2};Q_{O_2,O_3} \cap R_{O_1,O_3} \subseteq (P_{O_1,O_2} \cap R_{O_1,O_3};{Q_{O_2,O_3}}^{\mathsf{T}});(Q_{O_2,O_3} \cap {P_{O_1,O_2}}^{\mathsf{T}};R_{O_1,O_3})$

```
correctDedekindFormula
RF (RV (VarR "P" (OV (VarO "O_1")) (OV (VarO "O_2"))) :***: (RV (VarR "Q
" (OV (VarO "O_2")) (OV (VarO "O_3")))) :&&&: (RV (VarR "R" (OV (VarO
"O_1")) (OV (VarO "O_3")))) :<==: (RV (VarR "P" (OV (VarO "O_1")) (OV
(VarO "O_2"))) :&&&: (RV (VarR "R" (OV (VarO "O_1")) (OV (VarO "O_3"
))) :***: (Convs (RV (VarR "Q" (OV (VarO "O_2")) (OV (VarO "O_3")))))) :
***: (RV (VarR "Q" (OV (VarO "O_2")) (OV (VarO "O_3"))) :&&&: (Convs
(RV (VarR "P" (OV (VarO "O_1")) (OV (VarO "O_2")))) :***: (RV (VarR "R"
(OV (VarO "O_1")) (OV (VarO "O_3"))))))))
```

```
dedekindInASCII
P:***:Q:&&&:R :<==: P:&&&:R:***:TRAN(Q):***:Q:&&&:TRAN(P):***:R
```

firstOrderDedekindTeXSHORT
$\langle\forall e_1 : \langle\forall e_2 : ((\langle\exists e_3 : ((e_1,e_3) \in P) \wedge ((e_3,e_2) \in Q)\rangle) \wedge ((e_1,e_2) \in R)) \longrightarrow (\langle\exists e_3 : (((e_1,e_3) \in P) \wedge (\langle\exists e_4 : ((e_1,e_4) \in R) \wedge ((e_3,e_4) \in Q)\rangle)) \wedge (((e_3,e_2) \in Q) \wedge (\langle\exists e_4 : ((e_4,e_3) \in P) \wedge ((e_4,e_2) \in R)\rangle)\rangle)\rangle\rangle\rangle\rangle$

firstOrderDedekindTeXLONG
$\langle\forall e_{1(O_1)} \in O_1 : \langle\forall e_{2(O_3)} \in O_3 : ((\langle\exists e_{3(O_2)} \in O_2 : ((e_{1(O_1)},e_{3(O_2)}) \in P_{O_1,O_2}) \wedge ((e_{3(O_2)},e_{2(O_3)}) \in Q_{O_2,O_3})\rangle) \wedge ((e_{1(O_1)},e_{2(O_3)}) \in R_{O_1,O_3})) \longrightarrow (\langle\exists e_{3(O_2)} \in O_2 : (((e_{1(O_1)},e_{3(O_2)}) \in P_{O_1,O_2}) \wedge (\langle\exists e_{4(O_3)} \in O_3 : ((e_{1(O_1)},e_{4(O_3)}) \in R_{O_1,O_3}) \wedge ((e_{3(O_2)},e_{4(O_3)}) \in Q_{O_2,O_3})\rangle)) \wedge (((e_{3(O_2)},e_{2(O_3)}) \in Q_{O_2,O_3}) \wedge (\langle\exists e_{4(O_1)} \in O_1 : ((e_{4(O_1)},e_{3(O_2)}) \in P_{O_1,O_2}) \wedge ((e_{4(O_1)},e_{2(O_3)}) \in R_{O_1,O_3})\rangle)\rangle)\rangle\rangle\rangle\rangle$

The corresponding test is now installed for the Schröder rules.

```
schroederFormulae sI =
   let (_,_,_,rvs,_) = supply sI 0 0 0 3 0
       [a,b,c] = map RV rvs
       ab = a :***: b
       aTcBar = Convs a :***: (NegaR c)
       cBarbT = NegaR c :***: (Convs b)
       [sA,sB] = generalTypeOfFormulae
                    [SemEqu (RF (ab :<==: c)) (RF (aTcBar :<==: (NegaR b))),
                     SemEqu (RF (ab :<==: c)) (RF (cBarbT :<==: (NegaR a)))]
       [rSA,rSB]   = map renameSingle [sA,sB]
       firstOrderA = translateFormula (VU [] [] [] [] []) correctSchroederAFormula
       firstOrderB = translateFormula (VU [] [] [] [] []) correctSchroederBFormula
       firstOrderATeXLONG  = tEX True  firstOrderA
       isCorrSchroederAFirstOrder  = tEX True  firstOrderA
       isCorrSchroederBFirstOrder  = tEX True  firstOrderA
       firstOrderATeXSHORT = tEX False firstOrderA
       firstOrderBTeXLONG  = tEX True  firstOrderB
       firstOrderBTeXSHORT = tEX False firstOrderB
```

```
   in  (rSA,rSB,isWellFormed sA,isWellFormed rSB,
        tEX False rSA, tEX True rSA, tEX False rSB, tEX True rSB,
        stripFormula rSA,   stripFormula rSB, firstOrderA, firstOrderB,
        isWellFormed firstOrderA,isWellFormed firstOrderB,
        firstOrderATeXSHORT, firstOrderATeXLONG,
        firstOrderBTeXSHORT, firstOrderBTeXLONG)
```

Giving names to all these results occurs in the following assignment.

```
(correctSchroederAFormula,     correctSchroederBFormula,
 isCorrectSchroederA,          isCorrectSchroederB,
 teXSchroederAshort,           teXSchroederAlong,
 teXSchroederBshort,           teXSchroederBlong,
 schroederAInASCII,            schroederBInASCII,
 firstOrderSchroederA,         firstOrderSchroederB,
 isCorrectSchroederAFirstOrder,isCorrectSchroederBFirstOrder,
 firstOrderSchroederATeXSHORT, firstOrderSchroederATeXLONG,
 firstOrderSchroederBTeXSHORT, firstOrderSchroederBTeXLONG) = schroederFormulae 432
```

The results produced are as follows:

isCorrectSchroederA: True

teXSchroederAshort: $(P;Q \subseteq R) \longleftrightarrow (P^{\mathsf{T}};\overline{R} \subseteq \overline{Q})$

teXSchroederAlong
$(P_{O_1,O_2};Q_{O_2,O_3} \subseteq R_{O_1,O_3}) \longleftrightarrow ({P_{O_1,O_2}}^{\mathsf{T}};\overline{R_{O_1,O_3}} \subseteq \overline{Q_{O_2,O_3}})$

schroederAInASCII
P:***:Q :<==: R <===> TRAN(P):***:NEGA(R) :<==: NEGA(Q)

```
correctSchroederAFormula
SemEqu (RF (RV (VarR "P" (OV (VarO "O_1")) (OV (VarO "O_2"))) :***: (RV
(VarR "Q" (OV (VarO "O_2")) (OV (VarO "O_3")))) :<==: (RV (VarR "R" (OV
(VarO "O_1")) (OV (VarO "O_3")))))) (RF (Convs (RV (VarR "P" (OV (VarO "O
_1")) (OV (VarO "O_2")))) :***: (NegaR (RV (VarR "R" (OV (VarO "O_1"))
(OV (VarO "O_3"))))) :<==: (NegaR (RV (VarR "Q" (OV (VarO "O_2")) (OV (V
arO "O_3")))))))
```

isCorrectSchroederB: True

teXSchroederBshort
$(P;Q \subseteq R) \longleftrightarrow (\overline{R};Q^{\mathsf{T}} \subseteq \overline{P})$

teXSchroederBlong
$(P_{O_1,O_2};Q_{O_2,O_3} \subseteq R_{O_1,O_3}) \longleftrightarrow (\overline{R_{O_1,O_3}};{Q_{O_2,O_3}}^{\mathsf{T}} \subseteq \overline{P_{O_1,O_2}})$

schroederBInASCII

```
P:***:Q :<==: R <===> NEGA(R):***:TRAN(Q) :<==: NEGA(P)

correctSchroederBFormula
SemEqu (RF (RV (VarR "P" (OV (VarO "O_1")) (OV (VarO "O_2"))) :***: (RV
(VarR "Q" (OV (VarO "O_2")) (OV (VarO "O_3")))) :<==: (RV (VarR "R" (OV
(VarO "O_1")) (OV (VarO "O_3")))))) (RF (NegaR (RV (VarR "R" (OV (VarO "O
_1")) (OV (VarO "O_3")))) :***: (Convs (RV (VarR "Q" (OV (VarO "O_2"))
(OV (VarO "O_3"))))) :<==: (NegaR (RV (VarR "P" (OV (VarO "O_1")) (OV (V
arO "O_2")))))))

isCorrectSchroederAFirstOrder: True

firstOrderSchroederATeXSHORT
```

$(\langle\forall e_1 : \langle\forall e_2 : (\langle\exists e_3 : ((e_1, e_3) \in P) \wedge ((e_3, e_2) \in Q)\rangle) \longrightarrow ((e_1, e_2) \in R)\rangle\rangle) \longleftrightarrow (\langle\forall e_1 : \langle\forall e_2 : (\langle\exists e_3 : ((e_3, e_1) \in P) \wedge (\neg ((e_3, e_2) \in R))\rangle) \longrightarrow (\neg ((e_1, e_2) \in Q))\rangle\rangle\rangle)$

```
firstOrderSchroederATeXLONG
```

$(\langle\forall e_{1(O_1)} \in O_1 : \langle\forall e_{2(O_3)} \in O_3 : (\langle\exists e_{3(O_2)} \in O_2 : ((e_{1(O_1)}, e_{3(O_2)}) \in P_{O_1,O_2}) \wedge ((e_{3(O_2)}, e_{2(O_3)}) \in Q_{O_2,O_3})\rangle) \longrightarrow ((e_{1(O_1)}, e_{2(O_3)}) \in R_{O_1,O_3})\rangle\rangle) \longleftrightarrow (\langle\forall e_{1(O_2)} \in O_2 : \langle\forall e_{2(O_3)} \in O_3 : (\langle\exists e_{3(O_1)} \in O_1 : ((e_{3(O_1)}, e_{1(O_2)}) \in P_{O_1,O_2}) \wedge (\neg ((e_{3(O_1)}, e_{2(O_3)}) \in R_{O_1,O_3}))\rangle) \longrightarrow (\neg ((e_{1(O_2)}, e_{2(O_3)}) \in Q_{O_2,O_3}))\rangle\rangle\rangle)$

```
isCorrectSchroederBFirstOrder: True

firstOrderSchroederBTeXSHORT
```

$(\langle\forall e_1 : \langle\forall e_2 : (\langle\exists e_3 : ((e_1, e_3) \in P) \wedge ((e_3, e_2) \in Q)\rangle) \longrightarrow ((e_1, e_2) \in R)\rangle\rangle) \longleftrightarrow (\langle\forall e_1 : \langle\forall e_2 : (\langle\exists e_3 : (\neg ((e_1, e_3) \in R)) \wedge ((e_2, e_3) \in Q)\rangle) \longrightarrow (\neg ((e_1, e_2) \in P))\rangle\rangle\rangle)$

```
firstOrderSchroederBTeXLONG
```

$(\langle\forall e_{1(O_1)} \in O_1 : \langle\forall e_{2(O_3)} \in O_3 : (\langle\exists e_{3(O_2)} \in O_2 : ((e_{1(O_1)}, e_{3(O_2)}) \in P_{O_1,O_2}) \wedge ((e_{3(O_2)}, e_{2(O_3)}) \in Q_{O_2,O_3})\rangle) \longrightarrow ((e_{1(O_1)}, e_{2(O_3)}) \in R_{O_1,O_3})\rangle\rangle) \longleftrightarrow (\langle\forall e_{1(O_1)} \in O_1 : \langle\forall e_{2(O_2)} \in O_2 : (\langle\exists e_{3(O_3)} \in O_3 : (\neg ((e_{1(O_1)}, e_{3(O_3)}) \in R_{O_1,O_3})) \wedge ((e_{2(O_2)}, e_{3(O_3)}) \in Q_{O_2,O_3})\rangle) \longrightarrow (\neg ((e_{1(O_1)}, e_{2(O_2)}) \in P_{O_1,O_2}))\rangle\rangle\rangle)$

## 5.2 Characterization of Direct Sums

The direct sum is something like a disjoint union. To make this precise, a universal characterisation has been invented. Two category objects are bound together using two injective mappings $\iota, \kappa$ satisfying the following formulae, here presented in a long (via the subsequent `iotaInjectTotalTexEXS, kappaInjectTotalTexEXS, iotaKappaTTexEXS, iTiOrkTkOrEqualsIdTexEXS`)

$$\iota ; \iota^{\mathsf{T}} = \mathbb{I} \qquad \kappa ; \kappa^{\mathsf{T}} = \mathbb{I} \qquad \iota ; \kappa^{\mathsf{T}} \subseteq \perp\!\!\!\perp \qquad \iota^{\mathsf{T}} ; \iota \cup \kappa^{\mathsf{T}} ; \kappa = \mathbb{I}$$

as well as in a short form (via the subsequent `iotaInjectTotalTexEXL, kappaInjectTotalTexEXL, iotaKappaTTexEXL, iTiOrkTkOrEqualsIdTexEXL`):

$$\iota_{A,A+B} ; \iota_{A,A+B}{}^{\mathsf{T}} = \mathbb{I}_A \qquad \kappa_{B,A+B} ; \kappa_{B,A+B}{}^{\mathsf{T}} = \mathbb{I}_B \qquad \iota_{A,A+B} ; \kappa_{B,A+B}{}^{\mathsf{T}} \subseteq \perp\!\!\!\perp_{AB}$$
$$\iota_{A,A+B}{}^{\mathsf{T}} ; \iota_{A,A+B} \cup \kappa_{B,A+B}{}^{\mathsf{T}} ; \kappa_{B,A+B} = \mathbb{I}_{A+B}$$

```
sumCharacterizingFormulaeFor o1 o2 =
   let iota   = Iota  o1 o2
       kappa  = Kappa o1 o2
       iotaT  = Convs iota
       kappaT = Convs kappa
       iiT = iota   :***: iotaT
       kkT = kappa  :***: kappaT
       iTi = iotaT  :***: iota
       kTk = kappaT :***: kappa
       ikT = iota   :***: kappaT
       fs = generalTypeOfFormulae $ map (\f -> RF f)
          [iiT :===: Ident o1,                              --iotaInjectTotal
           kkT :===: Ident o2,                              --kappaInjectTotal
           ikT :<==: NullR o1 o2,                           --iotaKappaT
           iTi :|||: kTk :===: Ident (DirSum o1 o2)]        --iTiOrkTkOrEqualsId
       rfs = map renameSingle fs
       wff = formulaeAreWellFormed rfs
       sumFormulaeInTeXSHORT = map (makeTeXFormula False) rfs
       sumFormulaeInTeXLONG  = map (makeTeXFormula True ) rfs
       stripSumFormulae      = stripFormulae          rfs
       th = TH "Sum-Theory" [o1,o2] [] [] [] [] [] rfs
       sumTheoryWellformed   = checkTheoryWellDefined th
       translatedSumFormulae =
          translateFormulae (VU [] [] [] [] []) rfs
       translatedSumFormulaeToTeXSHORT =
          "$" ++ makeTeXFormulae False translatedSumFormulae ++ "$"
       translatedSumFormulaeToTeXLONG  =
          "$" ++ makeTeXFormulae True  translatedSumFormulae ++ "$"
       stripSumForm1stOrd = stripFormulae translatedSumFormulae
   in  (rfs,wff,sumFormulaeInTeXSHORT,sumFormulaeInTeXLONG,stripSumFormulae,
        translatedSumFormulae,
        translatedSumFormulaeToTeXSHORT,translatedSumFormulaeToTeXLONG,
        stripSumForm1stOrd,th,sumTheoryWellformed,
        iota, kappa, iiT, kkT,iTi,kTk)
```

Asking for wellformedness with `wellFormedSumCharacterizingFormulae` results in `True`. One may translate these formulae to first-order form resulting via `translatedSumFormulaeToTeXSHORT` into

$(\langle\forall e_1 : \langle\forall e_2 : (\langle\exists e_3 : ((e_1, e_3) \in \iota) \wedge ((e_2, e_3) \in \iota)\rangle) \longrightarrow (e_1 = e_2)\rangle\rangle) \wedge (\langle\forall e_1 : \langle\exists e_4 : (e_1, e_4) \in \iota\rangle\rangle)$,

$(\langle\forall e_1 : \langle\forall e_2 : (\langle\exists e_3 : ((e_1, e_3) \in \kappa) \wedge ((e_2, e_3) \in \kappa)\rangle) \longrightarrow (e_1 = e_2)\rangle\rangle) \wedge (\langle\forall e_1 : \langle\exists e_4 : (e_1, e_4) \in \kappa\rangle\rangle)$,

$\langle\forall e_1 : \langle\forall e_2 : \neg (\langle\exists e_5 : ((e_1, e_5) \in \iota) \wedge ((e_2, e_5) \in \kappa)\rangle)\rangle\rangle$,

$(\langle\forall e_1 : \langle\forall e_2 : ((\langle\exists e_3 : ((e_3, e_1) \in \iota) \wedge ((e_3, e_2) \in \iota)\rangle) \vee (\langle\exists e_3 : ((e_3, e_1) \in \kappa) \wedge ((e_3, e_2) \in \kappa)\rangle)) \longrightarrow (e_1 = e_2)\rangle\rangle) \wedge (\langle\forall e_1 : (\langle\exists e_4 : (e_4, e_1) \in \iota\rangle) \vee (\langle\exists e_4 : (e_4, e_1) \in \kappa\rangle)\rangle)$

or via `translatedSumFToTeXLONG`

$(\langle\forall e_{1(A)} \in A : \langle\forall e_{2(A)} \in A : (\langle\exists e_{3(A+B)} \in A + B : ((e_{1(A)}, e_{3(A+B)}) \in \iota_{A,A+B}) \wedge ((e_{2(A)}, e_{3(A+B)}) \in \iota_{A,A+B})\rangle) \longrightarrow (e_{1(A)} = e_{2(A)})\rangle\rangle) \wedge (\langle\forall e_{1(A)} \in A : \langle\exists e_{4(A+B)} \in A + B : (e_{1(A)}, e_{4(A+B)}) \in \iota_{A,A+B}\rangle\rangle)$,

$(\langle\forall e_{1(B)} \in B : \langle\forall e_{2(B)} \in B : (\langle\exists e_{3(A+B)} \in A + B : ((e_{1(B)}, e_{3(A+B)}) \in \kappa_{B,A+B}) \wedge ((e_{2(B)}, e_{3(A+B)}) \in \kappa_{B,A+B})\rangle) \longrightarrow (e_{1(B)} = e_{2(B)})\rangle\rangle) \wedge (\langle\forall e_{1(B)} \in B : \langle\exists e_{4(A+B)} \in A + B : (e_{1(B)}, e_{4(A+B)}) \in \kappa_{B,A+B}\rangle\rangle)$,

$\langle\forall e_{1(A)} \in A : \langle\forall e_{2(B)} \in B : \neg (\langle\exists e_{5(A+B)} \in A + B : ((e_{1(A)}, e_{5(A+B)}) \in \iota_{A,A+B}) \wedge ((e_{2(B)}, e_{5(A+B)}) \in \kappa_{B,A+B})\rangle)\rangle\rangle$,

$(\langle\forall e_{1(A+B)} \in A + B : \langle\forall e_{2(A+B)} \in A + B : ((\langle\exists e_{3(A)} \in A : ((e_{3(A)}, e_{1(A+B)}) \in \iota_{A,A+B}) \wedge ((e_{3(A)}, e_{2(A+B)}) \in \iota_{A,A+B})\rangle) \vee (\langle\exists e_{3(B)} \in B : ((e_{3(B)}, e_{1(A+B)}) \in \kappa_{B,A+B}) \wedge ((e_{3(B)}, e_{2(A+B)}) \in \kappa_{B,A+B})\rangle)) \longrightarrow (e_{1(A+B)} = e_{2(A+B)})\rangle\rangle) \wedge (\langle\forall e_{1(A+B)} \in A+B : (\langle\exists e_{4(A)} \in A : (e_{4(A)}, e_{1(A+B)}) \in \iota_{A,A+B}\rangle) \vee (\langle\exists e_{4(B)} \in B : (e_{4(B)}, e_{1(A+B)}) \in \kappa_{B,A+B}\rangle)\rangle)$

We decide for constant objects and names for these in the category objects and then instantiate the theory, followed by the definition of a corresponding model.

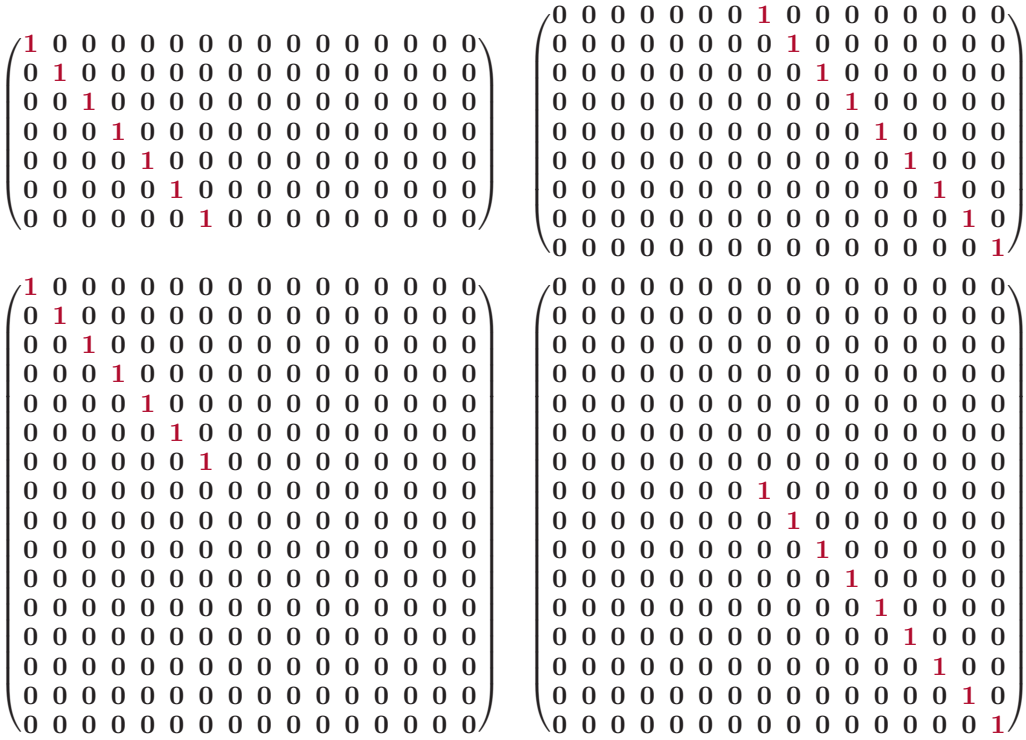
```
([co11,co22],_,_,_,_) = supplyConst ["A","B"] [] [] [] []
(sumCharFormulaeEX@[iotaInjectTotalEX,kappaInjectTotalEX,
 iotaKappaTEX,iTiOrkTkOrEqualsIdEX],sumFormulaeWellformed,
 sumFormulaeInTeXSHORT@[iotaInjectTotalTexEXS,kappaInjectTotalTexEXS,
 iotaKappaTTexEXS,iTiOrkTkOrEqualsIdTexEXS],
 sumFormulaeInTeXLONG@[iotaInjectTotalTexEXL,kappaInjectTotalTexEXL,
 iotaKappaTTexEXL,iTiOrkTkOrEqualsIdTexEXL],stripSumFormulae,
 translatedSumFormulae,translatedSumFToTeXSHORT,
 translatedSumFToTeXLONG,stripSumFormulaeFstOrd,
 sumTheoryEX,sumTheoryWellformed,
 iotaEX,kappaEX,iiTEX,kkTEX,iTiEX,kTkEX) =
    sumCharacterizingFormulaeFor (OC co11) (OC co22)

sumModelEX =
   MO "Sum-Model" sumTheoryEX
      [Carrier (OC co11) 7,Carrier (OC co22) 9] [] [] [] [] []  []
```

This model can be checked as to whether it is a model of the theory. It may also be used to compute the interpretations of the matrices important in this context.

```
isModelForSumTheory = checkIsModelForTheory sumModelEX

[iotaMatEX,kappaMatEX,iotaTIotaMatEX,kappaTKappaMatEX] =
   interpretRelaTerms sumModelEX ([],[],[]) [iotaEX,kappaEX,iTiEX,kTkEX]
```



Injections $\iota, \kappa$ and partial identities $\iota^{\mathsf{T}};\iota, \kappa^{\mathsf{T}};\kappa$ of a direct sum

```
sumFormulaeWellformed: True
```

```
stripSumFormulae
IOTA(A,A + B):***:TRAN(IOTA(A,A + B)) :===: IDEN, KAPPA(B,A + B):***:TRAN(
KAPPA(B,A + B)) :===: IDEN, IOTA(A,A + B):***:TRAN(KAPPA(B,A + B)) :<==:
NULL, TRAN(IOTA(A,A + B)):***:IOTA(A,A + B):|||:TRAN(KAPPA(B,A + B)):***:KA
PPA(B,A + B) :===: IDEN
```

It is also possible to first translate to first-order form and then to TEX, making formulae much clumsier:

```
translatedSumFToTeXSHORT
```

$(\langle\forall e_1 : \langle\forall e_2 : (\langle\exists e_3 : ((e_1, e_3) \in \iota) \wedge ((e_2, e_3) \in \iota)\rangle) \longrightarrow (e_1 = e_2)\rangle\rangle) \wedge (\langle\forall e_1 : \langle\exists e_4 : (e_1, e_4) \in \iota\rangle\rangle)$,

$(\langle\forall e_1 : \langle\forall e_2 : (\langle\exists e_3 : ((e_1, e_3) \in \kappa) \wedge ((e_2, e_3) \in \kappa)\rangle) \longrightarrow (e_1 = e_2)\rangle\rangle) \wedge (\langle\forall e_1 : \langle\exists e_4 : (e_1, e_4) \in \kappa\rangle\rangle)$,

$\langle\forall e_1 : \langle\forall e_2 : \neg (\langle\exists e_5 : ((e_1, e_5) \in \iota) \wedge ((e_2, e_5) \in \kappa)\rangle)\rangle\rangle$,

$(\langle\forall e_1 : \langle\forall e_2 : ((\langle\exists e_3 : ((e_3, e_1) \in \iota) \wedge ((e_3, e_2) \in \iota)\rangle) \vee (\langle\exists e_3 : ((e_3, e_1) \in \kappa) \wedge ((e_3, e_2) \in \kappa)\rangle)) \longrightarrow (e_1 = e_2)\rangle\rangle) \wedge (\langle\forall e_1 : (\langle\exists e_4 : (e_4, e_1) \in \iota\rangle) \vee (\langle\exists e_4 : (e_4, e_1) \in \kappa\rangle)\rangle)$

```
translatedSumFToTeXLONG
```

$(\langle\forall e_{1(A)} \in A : \langle\forall e_{2(A)} \in A : (\langle\exists e_{3(A+B)} \in A + B : ((e_{1(A)}, e_{3(A+B)}) \in \iota_{A,A+B}) \wedge ((e_{2(A)}, e_{3(A+B)}) \in \iota_{A,A+B})\rangle) \longrightarrow (e_{1(A)} = e_{2(A)})\rangle\rangle) \wedge (\langle\forall e_{1(A)} \in A : \langle\exists e_{4(A+B)} \in A + B : (e_{1(A)}, e_{4(A+B)}) \in \iota_{A,A+B}\rangle\rangle)$,

$(\langle\forall e_{1(B)} \in B : \langle\forall e_{2(B)} \in B : (\langle\exists e_{3(A+B)} \in A+B : ((e_{1(B)}, e_{3(A+B)}) \in \kappa_{B,A+B}) \wedge ((e_{2(B)}, e_{3(A+B)}) \in \kappa_{B,A+B})\rangle) \longrightarrow (e_{1(B)} = e_{2(B)})\rangle\rangle) \wedge (\langle\forall e_{1(B)} \in B : \langle\exists e_{4(A+B)} \in A+B : (e_{1(B)}, e_{4(A+B)}) \in \kappa_{B,A+B}\rangle\rangle)$,

$\langle\forall e_{1(A)} \in A : \langle\forall e_{2(B)} \in B : \neg (\langle\exists e_{5(A+B)} \in A+B : ((e_{1(A)}, e_{5(A+B)}) \in \iota_{A,A+B}) \wedge ((e_{2(B)}, e_{5(A+B)}) \in \kappa_{B,A+B})\rangle)\rangle\rangle$,

$(\langle\forall e_{1(A+B)} \in A+B : \langle\forall e_{2(A+B)} \in A+B : ((\langle\exists e_{3(A)} \in A : ((e_{3(A)}, e_{1(A+B)}) \in \iota_{A,A+B}) \wedge ((e_{3(A)}, e_{2(A+B)}) \in \iota_{A,A+B})\rangle) \vee (\langle\exists e_{3(B)} \in B : ((e_{3(B)}, e_{1(A+B)}) \in \kappa_{B,A+B}) \wedge ((e_{3(B)}, e_{2(A+B)}) \in \kappa_{B,A+B})\rangle)) \longrightarrow (e_{1(A+B)} = e_{2(A+B)})\rangle\rangle) \wedge (\langle\forall e_{1(A+B)} \in A+B : (\langle\exists e_{4(A)} \in A : (e_{4(A)}, e_{1(A+B)}) \in \iota_{A,A+B}\rangle) \vee (\langle\exists e_{4(B)} \in B : (e_{4(B)}, e_{1(A+B)}) \in \kappa_{B,A+B}\rangle)\rangle)$

## 5.3 Characterization of Constructed Injections

When a vector term is given, one sometimes needs the corresponding injection relation. This enforces to give a construction of the domain of this injection.

```
injConstructCharacterizingFormulae vt =
   let domainI = InjFrom vt
       injectionI = InjTerm vt
       injectionIT  = Convs injectionI
       iiTRAW = injectionI  :***: injectionIT
       iTiRAW = injectionIT :***: injectionI
       [iiT,iTi] = generalTypeOfRelaTerms [iiTRAW,iTiRAW]
   in  (domainI,
        iiT :===: Ident domainI,
        iTi :<==: Ident (domVT vt),
        isWellFormed iiT,
        isWellFormed iTi)
```

## 5.4 Characterization of Direct Products

In a closely related form also direct products may be formed. To this end two surjective mappings $\pi, \rho$ are used satisfying in a long or short form

```
prodCharacterizingFormulaeFor o1 o2 =
   let ppi = Pi  o1 o2
       rho = Rho o1 o2
       ppiT = Convs ppi
       rhoT = Convs rho
       ppiTppi = ppiT :***: ppi
       rhoTrho = rhoT :***: rho
       ppippiT = ppi  :***: ppiT
       rhorhoT = rho  :***: rhoT
       fs = generalTypeOfFormulae $ map (\f -> RF f)
           [ppiTppi  :===: Ident o1,
            rhoTrho  :===: Ident o2,
```

```
            UnivR o1 o2 :<==: (ppiT :***: rho),
            ppippiT :&&&: rhorhoT :===: Ident (DirPro o1 o2)]
       rfs = map renameSingle fs
       wff = formulaeAreWellFormed rfs
       productFormulaeInTeXSHORT = makeTeXFormulae False rfs
       productFormulaeInTeXLONG  = makeTeXFormulae True  rfs
       stripProductFormulae = stripFormulae rfs
       translatedProductFormulae = translateFormulae (VU [] [] [] [] []) rfs
       translatedProductFormulaeToTeXSHORT =
          "$" ++ (makeTeXFormulae False translatedProductFormulae) ++ "$"
       translatedProductFormulaeToTeXLONG =
          "$" ++ (makeTeXFormulae True  translatedProductFormulae) ++ "$"
       wellFormedProductTheory =
          checkTheoryWellDefined th
       th =         TH "Product-Theory" [o1,o2] [] [] [] [] [] [] rfs
       stripProductFormulaeFstOrd = stripFormulae translatedProductFormulae
   in  (rfs,wff,productFormulaeInTeXSHORT,
        productFormulaeInTeXLONG,
        stripProductFormulae,
        translatedProductFormulae,
        translatedProductFormulaeToTeXSHORT,
        translatedProductFormulaeToTeXLONG,
        stripProductFormulaeFstOrd,
        th,wellFormedProductTheory,
        ppi,rho,ppippiT,rhorhoT)


([co1111,co2222],_,_,_,_) = supplyConst ["A","B"] [] [] [] []
(correctProdCharFormulaeEX@[piUnivSurjEX,rhoUnivSurjEX,
                     piTrhoEX,piPiTANDRhoRhoTEqualsIdEX],
 areWffproductFormulae,productFormulaeInTeXSHORT,
 productFormulaeInTeXLONG,
 stripProductFormulae,
 translatedProdFormulaeFstOrd,
 transltdProdFormulaeTeXSHORT,
 transltdProdFormulaeTeXLONG,
 stripProductFormulaeFstOrd,
 productTheoryEX, wellFormedProductTheory,
 piEX,rhoEX,piPiTEX,rhoRhoTEX) =
    prodCharacterizingFormulaeFor (OC co1111) (OC co2222)



productModelEX =
   MO "Product-Model" productTheoryEX [Carrier (OC co1111) 3,Carrier (OC co2222) 5]
      [] [] [] [] [] []
isModelForProductTheory =
   checkIsModelForTheory productModelEX
[piMatEX,rhoMatEX,piPiTMatEX,rhoRhoTMatEX] =
   interpretRelaTerms productModelEX ([],[],[]) [piEX,rhoEX,piPiTEX,rhoRhoTEX]
```

```
correctProdCharFormulaeEX
[RF (Convs (Pi (OC (CstO "A")) (OC (CstO "B"))) :***: (Pi (OC (CstO "A"))
(OC (CstO "B"))) :===: (Ident (OC (CstO "A")))),RF (Convs (Rho (OC (CstO "
A")) (OC (CstO "B"))) :***: (Rho (OC (CstO "A")) (OC (CstO "B"))) :===: (
Ident (OC (CstO "B")))),RF (UnivR (OC (CstO "A")) (OC (CstO "B")) :<==: (C
onvs (Pi (OC (CstO "A")) (OC (CstO "B"))) :***: (Rho (OC (CstO "A")) (OC (
CstO "B"))))),RF (Pi (OC (CstO "A")) (OC (CstO "B")) :***: (Convs (Pi (OC
(CstO "A")) (OC (CstO "B")))) :&&&: (Rho (OC (CstO "A")) (OC (CstO "B")
) :***: (Convs (Rho (OC (CstO "A")) (OC (CstO "B"))))) :===: (Ident (DirP
ro (OC (CstO "A")) (OC (CstO "B")))))]

areWffproductFormulae: True

piUnivSurjEX
RF (Convs (Pi (OC (CstO "A")) (OC (CstO "B"))) :***: (Pi (OC (CstO "A")) (
OC (CstO "B"))) :===: (Ident (OC (CstO "A"))))

rhoUnivSurjEX
RF (Convs (Rho (OC (CstO "A")) (OC (CstO "B"))) :***: (Rho (OC (CstO "A"))
(OC (CstO "B"))) :===: (Ident (OC (CstO "B"))))

piTrhoEX
RF (UnivR (OC (CstO "A")) (OC (CstO "B")) :<==: (Convs (Pi (OC (CstO "A"))
(OC (CstO "B"))) :***: (Rho (OC (CstO "A")) (OC (CstO "B")))))

piPiTANDRhoRhoTEqualsIdEX
RF (Pi (OC (CstO "A")) (OC (CstO "B")) :***: (Convs (Pi (OC (CstO "A")) (O
C (CstO "B")))) :&&&: (Rho (OC (CstO "A")) (OC (CstO "B")) :***: (Conv
s (Rho (OC (CstO "A")) (OC (CstO "B"))))) :===: (Ident (DirPro (OC (CstO "
A")) (OC (CstO "B")))))

productFormulaeInTeXSHORT
```

$$\pi^{\mathsf{T}};\pi = \mathbb{I},$$

$$\rho^{\mathsf{T}};\rho = \mathbb{I},$$

$$\mathbb{T} \subseteq \pi^{\mathsf{T}};\rho,$$

$$\pi;\pi^{\mathsf{T}} \cap \rho;\rho^{\mathsf{T}} = \mathbb{I}$$

```
productFormulaeInTeXLONG
```

$$\pi_{A\times B,A}{}^{\mathsf{T}};\pi_{A\times B,A} = \mathbb{I}_A,$$

$$\rho_{A\times B,B}{}^{\mathsf{T}};\rho_{A\times B,B} = \mathbb{I}_B,$$

$$\mathbb{T}_{AB} \subseteq \pi_{A\times B,A}{}^{\mathsf{T}};\rho_{A\times B,B},$$

$$\pi_{A\times B,A};\pi_{A\times B,A}{}^{\mathsf{T}} \cap \rho_{A\times B,B};\rho_{A\times B,B}{}^{\mathsf{T}} = \mathbb{I}_{A\times B}$$

```
transltdProdFormulaeTeXLONG
```

$(\langle\forall e_{1(A)} \in A : \langle\forall e_{2(A)} \in A : (\langle\exists e_{3(A\times B)} \in A \times B : ((e_{3(A\times B)}, e_{1(A)}) \in \pi_{A\times B,A}) \wedge ((e_{3(A\times B)}, e_{2(A)}) \in \pi_{A\times B,A})\rangle) \longrightarrow (e_{1(A)} = e_{2(A)})\rangle\rangle) \wedge (\langle\forall e_{1(A)} \in A : \langle\exists e_{4(A\times B)} \in A \times B : (e_{4(A\times B)}, e_{1(A)}) \in \pi_{A\times B,A}\rangle\rangle)$,

$(\langle\forall e_{1(B)} \in B : \langle\forall e_{2(B)} \in B : (\langle\exists e_{3(A\times B)} \in A \times B : ((e_{3(A\times B)}, e_{1(B)}) \in \rho_{A\times B,B}) \wedge ((e_{3(A\times B)}, e_{2(B)}) \in \rho_{A\times B,B})\rangle) \longrightarrow (e_{1(B)} = e_{2(B)})\rangle\rangle) \wedge (\langle\forall e_{1(B)} \in B : \langle\exists e_{4(A\times B)} \in A \times B : (e_{4(A\times B)}, e_{1(B)}) \in \rho_{A\times B,B}\rangle\rangle)$,

$\langle\forall e_{1(A)} \in A : \langle\forall e_{2(B)} \in B : \langle\exists e_{5(A\times B)} \in A \times B : ((e_{5(A\times B)}, e_{1(A)}) \in \pi_{A\times B,A}) \wedge ((e_{5(A\times B)}, e_{2(B)}) \in \rho_{A\times B,B})\rangle\rangle\rangle$,

$(\langle\forall e_{1(A\times B)} \in A \times B : \langle\forall e_{2(A\times B)} \in A \times B : ((\langle\exists e_{3(A)} \in A : ((e_{1(A\times B)}, e_{3(A)}) \in \pi_{A\times B,A}) \wedge ((e_{2(A\times B)}, e_{3(A)}) \in \pi_{A\times B,A})\rangle) \wedge (\langle\exists e_{3(B)} \in B : ((e_{1(A\times B)}, e_{3(B)}) \in \rho_{A\times B,B}) \wedge ((e_{2(A\times B)}, e_{3(B)}) \in \rho_{A\times B,B})\rangle)) \longrightarrow (e_{1(A\times B)} = e_{2(A\times B)})\rangle\rangle) \wedge (\langle\forall e_{1(A\times B)} \in A\times B : (\langle\exists e_{4(A)} \in A : (e_{1(A\times B)}, e_{4(A)}) \in \pi_{A\times B,A}\rangle) \wedge (\langle\exists e_{4(B)} \in B : (e_{1(A\times B)}, e_{4(B)}) \in \rho_{A\times B,B}\rangle)\rangle)$

```
stripProductFormulae
TRAN(PI(A x B,A)):***:PI(A x B,A) :===: IDEN, TRAN(RHO(A x B,B)):***:RHO(A
x B,B) :===: IDEN, UNIV :<==: TRAN(PI(A x B,A)):***:RHO(A x B,B), PI(A x
B,A):***:TRAN(PI(A x B,A)):&&&:RHO(A x B,B):***:TRAN(RHO(A x B,B)) :===
: IDEN

stripProductFormulaeFstOrd
FORALL e_1: FORALL e_2: EXISTS e_3: PI(A x B,A)( e_3 , e_1 ) AND PI
(A x B,A)( e_3 , e_2 ) ===> e_1 = e_2 AND FORALL e_1: EXISTS e_4
: PI(A x B,A)( e_4 , e_1 ) AND PI(A x B,A)( e_4 , e_1 ), FORALL e_1:
FORALL e_2: EXISTS e_3: RHO(A x B,B)( e_3 , e_1 ) AND RHO(A x B,B)(
e_3 , e_2 ) ===> e_1 = e_2 AND FORALL e_1: EXISTS e_4: RHO(A x
B,B)( e_4 , e_1 ) AND RHO(A x B,B)( e_4 , e_1 ), FORALL e_1: FORALL e
_2: EXISTS e_5: PI(A x B,A)( e_5 , e_1 ) AND RHO(A x B,B)( e_5 , e_
2 ), FORALL e_1: FORALL e_2: EXISTS e_3: PI(A x B,A)( e_1 , e_3 ) A
ND PI(A x B,A)( e_2 , e_3 ) AND EXISTS e_3: RHO(A x B,B)( e_1 , e_3 )
AND RHO(A x B,B)( e_2 , e_3 ) ===> e_1 = e_2 AND FORALL e_1: EXI
STS e_4: PI(A x B,A)( e_1 , e_4 ) AND PI(A x B,A)( e_1 , e_4 ) AND EX
ISTS e_4: RHO(A x B,B)( e_1 , e_4 ) AND RHO(A x B,B)( e_1 , e_4 )
```

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

Projections $\pi, \rho$ together with equivalences $\pi;\pi^{\mathsf{T}}, \rho;\rho^{\mathsf{T}}$

Also printing in ASCII-form is possible, namely

```
TRAN(PI(O1 x O2,O1)):***:PI(O1 x O2,O1) :===: IDEN,
TRAN(RHO(O1 x O2,O2)):***:RHO(O1 x O2,O2) :===: IDEN,
UNIV :<==: TRAN(PI(O1 x O2,O1)):***:RHO(O1 x O2,O2),
PI(O1 x O2,O1):***:TRAN(PI(O1 x O2,O1)):&&&:RHO(O1 x O2,O2):***:
TRAN(RHO(O1 x O2,O2)) :===: IDEN
```

The same formulae are now first translated to first-order form and then automatically to TEX. Further development will certainly improve the rather stupid form generated here.

$$\langle\forall av_1 :\ \langle\forall av_2 :\ \langle\exists av_3 :\ (av_3, av_1) \in \pi\ \wedge\ (av_3, av_2) \in \pi\rangle\ \Longrightarrow\ av_1 = av_2\rangle\rangle\ \wedge \\ \langle\forall av_1 :\ \langle\exists av_4 :\ (av_4, av_1) \in \pi\rangle\rangle,$$

$$\langle\forall av_1 :\ \langle\forall av_2 :\ \langle\exists av_3 :\ (av_3, av_1) \in \rho\ \wedge\ (av_3, av_2) \in \rho\rangle\ \Longrightarrow\ av_1 = av_2\rangle\rangle\ \wedge \\ \langle\forall av_1 :\ \langle\exists av_4 :\ (av_4, av_1) \in \rho\rangle\rangle,$$

$$\langle\forall av_1 :\ \langle\forall av_2 :\ \langle\exists av_3 :\ (av_3, av_1) \in \pi\ \wedge\ (av_3, av_2) \in \rho\rangle\rangle\rangle,$$

$$\langle\forall av_1 :\ \langle\forall av_2 :\ \langle\exists av_3 :\ (av_1, av_3) \in \pi\ \wedge\ (av_2, av_3) \in \pi\rangle\ \wedge \\ \langle\exists av_3 :\ (av_1, av_3) \in \rho\ \wedge\ (av_2, av_3) \in \rho\rangle\ \Longrightarrow\ av_1 = av_2\rangle\rangle\ \wedge \\ \langle\forall av_1 :\ \langle\exists av_4 :\ (av_1, av_4) \in \pi\rangle\ \wedge\ \langle\exists av_4 :\ (av_1, av_4) \in \rho\rangle\rangle$$

## 5.5 Characterization of Direct Powers

Yet another universally characterized construct is the direct power. It models the is_element_of relation between a set $X$ and its powerset $\mathcal{P}(X)$. We model this with a relation $\epsilon$ satisfying

$$\mathtt{syq}(\epsilon,\epsilon);\mathtt{syq}(\epsilon,\epsilon)^{\mathsf{T}} \subseteq \mathbb{I},\quad \forall v : \langle\mathbb{T} \subseteq \mathbb{T};\mathtt{syq}(\epsilon, v)\rangle$$

$$\mathtt{syq}(\epsilon_{O_1},\epsilon_{O_1});\mathtt{syq}(\epsilon_{O_1},\epsilon_{O_1})^{\mathsf{T}} \subseteq \mathbb{I}_{\mathcal{P}(O_1)},\ \forall v \subseteq O_1 : \langle\mathbb{T} \subseteq \mathbb{T};\mathtt{syq}(\epsilon_{O_1}, v)\rangle$$

```
powerCharacterizingFormulae o1 =
   let epsiTe = Epsi o1
       eee  = VarE "e" o1
       vvv  = VarV "v" o1
       al   = UnivR UnitOb (DirPow o1)
       syQepsiepsi = SyQ epsiTe epsiTe
       syQv vv = Syq epsiTe vv
       powerFormulae = generalTypeOfFormulae
          [RF (syQepsiepsi :<==: (Ident (DirPow o1))),
           VF (QuantVectForm Univ vvv
                [VF (UnivV UnitOb :<===: (al :****: (syQv (VV vvv))))])]
       areWellFormedPowerFormulae = and $ map isWellFormed powerFormulae
       powerFormulaeInTeXLONG  = map (makeTeXFormula True)  powerFormulae
       powerFormulaeInTeXSHORT = map (makeTeXFormula False) powerFormulae
       stripPowerFormulae      = stripFormulae powerFormulae
       translatedPowerFormulaOnlyFirst =
          translateFormula (VU [] [] [] [] []) $ head powerFormulae
       translPowerFormula1ToTeXLONG    =
          makeTeXFormula True  translatedPowerFormulaOnlyFirst
       translPower1FormulaToTeXSHORT   =
```

```
          makeTeXFormula False translatedPowerFormulaOnlyFirst
       powerTheory = TH "Power-Theory" [o1] [] [] [] [] [] [] powerFormulae
       wellFormedPowerTheory = checkTheoryWellDefined powerTheory
       singleton = epsiTe :&&&:
                   (NegaR ((NegaR (Ident $ domRT epsiTe)) :***: epsiTe))
       convertVTtoET = \vt -> ThatV $ Syq epsiTe vt
       convertETtoVT = \et -> epsiTe :****: (PointVect et)
   in  (powerFormulae,areWellFormedPowerFormulae,
        powerFormulaeInTeXSHORT,powerFormulaeInTeXLONG,stripPowerFormulae,
        translatedPowerFormulaOnlyFirst,
        translPower1FormulaToTeXSHORT,translPowerFormula1ToTeXLONG,
        powerTheory,wellFormedPowerTheory,epsiTe,singleton,
        (vvv,convertVTtoET),(eee,convertETtoVT))

([co111],_,_,_,_) = supplyConst ["A"] [] [] [] []
(correctPowerFormulae,areWellFormedPowerFormulae,
 powerFormulaeInTeXSHORT@[epsilonInjectiveSHORT,epsilonContainsAllColumnsSHORT],
 powerFormulaeInTeXLONG@[epsilonInjectiveLONG,epsilonContainsAllColumnsLONG],
 stripPowerFormulae,
 transltdPowFormulaOnlyFirst,
 translPowFOFormulaToTeXSHORT,translPowFOFormulaToTeXLONG,
 powerTheoryEX,powerTheoryEXIsWellFormed,epsiTerm,singletonTerm,
 (varConvertVTtoET,convertVTtoET),
 (varConvertETtoVT,convertETtoVT)) = powerCharacterizingFormulae (OC co111)


(powerModelEX,epsiMat,singletonMat,convFctVTET,convFctETVT) =
   let powerModelEX = MO "Power-Model" powerTheoryEX
                         [Carrier (OC co111) 5] [] [] [] [] [] []
       epsiMat1 = interpretRelaTerm powerModelEX ([],[],[]) epsiTerm
       singletonMat1 = interpretRelaTerm powerModelEX ([],[],[]) singletonTerm
       ccc v = interpretElemTerm powerModelEX
               ([],[(varConvertVTtoET,v)],[]) (convertVTtoET $ VV varConvertVTtoET)
       ddd w = interpretVectTerm powerModelEX
                ([(varConvertETtoVT,w)],[],[]) (convertETtoVT $ EV varConvertETtoVT)
   in  (powerModelEX,epsiMat1,singletonMat1,ccc,ddd)
isModelForPowerTheory = checkIsModelForTheory powerModelEX
```

What we need in addition is a singleton injection from the base set $X$ into $\mathcal{P}(X)$. Here, an element term will be converted to a vector term over the power set.

```
singletonSet et =
   let d = domET et
       th = powerTheoryEX
   in  Syq (Epsi d) (PointVect et)
wellFormedSingletonSet ev =
```

```
   vectTermIsWellFormed $ generalTypeOfVectTerm $ singletonSet $ EV ev
testPowSingleton =
   let ([cc1],_,_,_,_) = supply 77 1 0 0 0 0
       ov = OV cc1
       ev = VarE "e" ov
       et = EV ev
   in  interpretVectTerm powerModelEX ([(ev, 3)],[],[]) (singletonSet et)
```

epsilonInjectiveSHORT
$\mathtt{syq}(\epsilon, \epsilon) \subseteq \mathbb{I}$

epsilonContainsAllColumnsSHORT
$\langle \forall v : \mathbb{T} \subseteq \mathbb{T}; \mathtt{syq}(\epsilon, v) \rangle$

epsilonInjectiveLONG
$\mathtt{syq}(\epsilon_A, \epsilon_A) \subseteq \mathbb{I}_{\mathcal{P}(A)}$

epsilonContainsAllColumnsLONG
$\langle \forall v_A \subseteq A : \mathbb{T}_{\mathbb{I}} \subseteq \mathbb{T}_{\mathbb{I}_{\mathcal{P}(A)}}; \mathtt{syq}(\epsilon_A, v_A) \rangle$

```
correctPowerFormulae
[RF (SyQ (Epsi (OC (CstO "A"))) (Epsi (OC (CstO "A"))) :<==: (Ident (DirPo
w (OC (CstO "A"))))),VF (QuantVectForm Univ (VarV "v" (OC (CstO "A"))) [VF
(UnivV UnitOb :<===: (UnivR UnitOb (DirPow (OC (CstO "A"))) :****: (Syq (
Epsi (OC (CstO "A"))) (VV (VarV "v" (OC (CstO "A")))))))])]

areWellFormedPowerFormulae
True

stripPowerFormulae
SyQ(EPSI(A),EPSI(A)) :<==: IDEN, FORALL v: UNIV :<===: UNIV:****:Syq(EPS
I(A),v)

powerTheoryEX
TH "Power-Theory" [OC (CstO "A")] [] [] [] [] [] [RF (SyQ (Epsi (OC (CstO "
A"))) (Epsi (OC (CstO "A"))) :<==: (Ident (DirPow (OC (CstO "A"))))),VF (Q
uantVectForm Univ (VarV "v" (OC (CstO "A"))) [VF (UnivV UnitOb :<===: (Uni
vR UnitOb (DirPow (OC (CstO "A"))) :****: (Syq (Epsi (OC (CstO "A"))) (VV
(VarV "v" (OC (CstO "A")))))))])]

powerTheoryEXIsWellFormed
True
```

translPowFOFormulaToTeXLONG
$\langle \forall e_{1(\mathcal{P}(A))} \in \mathcal{P}(A) : \langle \forall e_{2(\mathcal{P}(A))} \in \mathcal{P}(A) : ((\neg (\langle \exists e_{3(A)} \in A : ((e_{3(A)}, e_{1(\mathcal{P}(A))}) \in \epsilon_A) \wedge (\neg ((e_{3(A)}, e_{2(\mathcal{P}(A))}) \in \epsilon_A))\rangle)) \wedge (\neg (\langle \exists e_{3(A)} \in A : (\neg ((e_{3(A)}, e_{1(\mathcal{P}(A))}) \in \epsilon_A)) \wedge ((e_{3(A)}, e_{2(\mathcal{P}(A))}) \in \epsilon_A)\rangle))) \longrightarrow (e_{1(\mathcal{P}(A))} = e_{2(\mathcal{P}(A))})\rangle\rangle$

epsiMat

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

```
singletonMat
```

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

## 5.6 Characterization of Parallel Products

Yet another universally characterized construct is the partial product, which depends on a relation $\epsilon$ satisfying

```
partProductCharacterizingFormulae =
   let ([co1,co2],_,_,_,_) = supply 666 2 0 0 0 0
       cc1   = OV co1
       cc2   = OV co2
       ppi = PPi  (ParObj cc1) (ParObj cc2)
       rho = PRho (ParObj cc1) (ParObj cc2)
       ppiT = TranspP ppi
       rhoT = TranspP rho
       ppiTppi = ppiT :*****: ppi
       rhoTrho = rhoT :*****: rho
       ppippiT = ppi  :*****: ppiT
       rhorhoT = rho  :*****: rhoT
   in  map (\f -> PF f)
       [ppiTppi  :<====: IdentP (ParObj cc1),
        IdentP (ParObj cc1) :<====: ppiTppi,
        rhoTrho  :<====: IdentP (ParObj cc2),
        IdentP (ParObj cc2) :<====: rhoTrho,
        ppiT :*****: rho  :<====:  (NullP (ParObj cc1) (ParObj cc2)),
        --StrictPartContained (ppippiT :|||||: rhorhoT)
                      --(IdentP (ParPro (ParObj cc1) (ParObj cc2))),
        IdentP (ParPro (ParObj cc1) (ParObj cc2))
                                       :<====: (ppippiT :|||||: rhorhoT)]
wellFormedPartProductCharacterizingFormulae =
   formulaeAreWellFormed partProductCharacterizingFormulae
```

## 5.7 Test Actualization

In the course of the development of this language frequent changes occurred all over again. In case we had included test results in the text, these would then usually be no longer up-to-date. So we have chosen the following technique. Test results in the report are always TEX-macros. These macros are generated to a file by one big test run which contains all the tests.

```
showP pt = "\\vbox{\\hbox{\\tt " ++
           ((cutInPieces . prefixSlashToSpecChars) (show pt)) ++ "}}"
showS st = "\\vbox{\\hbox{\\tt " ++
```

```
           (cutInPieces1. prefixSlashToSpecChars1)       st ++ "}}"

testDruckTestResultMacros =
   let printFile = "SHUTTLE:RelaHaRes:TestOrdner:TestResultMacros"
       printTT x y = "\\long\\def\\" ++ x ++ "{"       ++ showP y ++  "}\n\n\n\n"
       printTS x y = "\\long\\def\\" ++ x ++ "{"       ++ showS y ++  "}\n\n\n\n"
       prinTeX x y = "\\long\\def\\" ++ x ++ "{"       ++       y ++  "}\n\n\n\n"
       priDTeX x y = "\\long\\def\\" ++ x ++ "{$"      ++       y ++ "$}\n\n\n\n"
       prinMat x y = "\\long\\def\\" ++ x ++ "{\\footnotesize{$\\tt "
                     ++ druckTeXMatH y ++ "$}}\n\n\n\n"
   in  do writeFile  printFile
           ("\n\n\\bigskip\n\\noindent\n" ++
           printTT "correctDedekindFormula"         correctDedekindFormula          ++
           priDTeX "teXDedekindlong"                teXDedekindlong                 ++
           priDTeX "teXDedekindshort"               teXDedekindshort                ++
           printTS "dedekindInASCII"                dedekindInASCII                 ++
           printTT "isCorrectDedekind"              isCorrectDedekind               ++
           printTT "firstOrderDedekind"             firstOrderDedekind              ++
           priDTeX "firstOrderDedekindTeXLONG"      firstOrderDedekindTeXLONG       ++
           priDTeX "firstOrderDedekindTeXSHORT"     firstOrderDedekindTeXSHORT      ++
                     -- Ende DedekindTestActualResult
           printTT "correctSchroederAFormula"       correctSchroederAFormula        ++
           printTT "correctSchroederBFormula"       correctSchroederBFormula        ++
           priDTeX "teXSchroederAlong"              teXSchroederAlong               ++
           priDTeX "teXSchroederBlong"              teXSchroederBlong               ++
           priDTeX "teXSchroederAshort"             teXSchroederAshort              ++
           priDTeX "teXSchroederBshort"             teXSchroederBshort              ++
           printTS "schroederAInASCII"              schroederAInASCII               ++
           printTS "schroederBInASCII"              schroederBInASCII               ++
           printTT "isCorrectSchroederA"            isCorrectSchroederA             ++
           printTT "isCorrectSchroederB"            isCorrectSchroederB             ++
           printTT "firstOrderSchroederA"           firstOrderSchroederA            ++
           printTT "firstOrderSchroederB"           firstOrderSchroederB            ++
           printTT "isCorrectSchroederAFirstOrder"  isCorrectSchroederAFirstOrder   ++
           printTT "isCorrectSchroederBFirstOrder"  isCorrectSchroederBFirstOrder   ++
           priDTeX "firstOrderSchroederATeXLONG"    firstOrderSchroederATeXLONG     ++
           priDTeX "firstOrderSchroederATeXSHORT"   firstOrderSchroederATeXSHORT    ++
           priDTeX "firstOrderSchroederBTeXLONG"    firstOrderSchroederBTeXLONG     ++
           priDTeX "firstOrderSchroederBTeXSHORT"   firstOrderSchroederBTeXSHORT    ++
                     -- Ende SchroederTestActualResults
           priDTeX "iotaInjectTotalTexEXS"          iotaInjectTotalTexEXS           ++
           priDTeX "kappaInjectTotalTexEXS"         kappaInjectTotalTexEXS          ++
           priDTeX "iotaKappaTTexEXS"               iotaKappaTTexEXS                ++
           priDTeX "iTiOrkTkOrEqualsIdTexEXS"       iTiOrkTkOrEqualsIdTexEXS        ++
           priDTeX "iotaInjectTotalTexEXL"          iotaInjectTotalTexEXL           ++
           priDTeX "kappaInjectTotalTexEXL"         kappaInjectTotalTexEXL          ++
           priDTeX "iotaKappaTTexEXL"               iotaKappaTTexEXL                ++
           priDTeX "iTiOrkTkOrEqualsIdTexEXL"       iTiOrkTkOrEqualsIdTexEXL        ++
           printTS "stripSumFormulae"               stripSumFormulae                ++
           printTT "sumFormulaeWellformed"          sumFormulaeWellformed           ++
```

```
 prinTeX "translatedSumFToTeXLONG"        translatedSumFToTeXLONG         ++
 prinMat "iotaMatEX"                      iotaMatEX                       ++
 prinMat "kappaMatEX"                     kappaMatEX                      ++
 prinMat "iotaTIotaMatEX"                 iotaTIotaMatEX                  ++
 prinMat "kappaTKappaMatEX"               kappaTKappaMatEX                ++
 printTT "translatedSumFormulae"          translatedSumFormulae           ++
 printTS "stripSumFormulaeFstOrd"         stripSumFormulaeFstOrd          ++
 prinTeX "translatedSumFToTeXSHORT"       translatedSumFToTeXSHORT        ++
           -- Ende SumTestActualResults
 printTT "correctPowerFormulae"           correctPowerFormulae            ++
 priDTeX "epsilonInjectiveSHORT"          epsilonInjectiveSHORT           ++
 priDTeX "epsilonContainsAllColumnsSHORT" epsilonContainsAllColumnsSHORT  ++
 priDTeX "epsilonInjectiveLONG"           epsilonInjectiveLONG            ++
 priDTeX "epsilonContainsAllColumnsLONG"  epsilonContainsAllColumnsLONG   ++
 printTS "stripPowerFormulae"             stripPowerFormulae              ++
 printTT "areWellFormedPowerFormulae"     areWellFormedPowerFormulae      ++
 printTT "powerTheoryEX"                  powerTheoryEX                   ++
 printTT "powerTheoryEXIsWellFormed"      powerTheoryEXIsWellFormed       ++
 printTT "transltdPowFormulaOnlyFirst"    transltdPowFormulaOnlyFirst     ++
 priDTeX "translPowFOFormulaToTeXSHORT"   translPowFOFormulaToTeXSHORT    ++
 prinMat "epsiMat"                        epsiMat                         ++
 prinMat "singletonMat"                   singletonMat                    ++
 priDTeX "translPowFOFormulaToTeXLONG"    translPowFOFormulaToTeXLONG     ++
           -- Ende PowTestActualResults
 printTT "correctProdCharFormulaeEX"      correctProdCharFormulaeEX       ++
 printTT "areWffproductFormulae"          areWffproductFormulae           ++
 printTT "piUnivSurjEX"                   piUnivSurjEX                    ++
 printTT "rhoUnivSurjEX"                  rhoUnivSurjEX                   ++
 printTT "piTrhoEX"                       piTrhoEX                        ++
 printTT "piPiTANDRhoRhoTEqualsIdEX"      piPiTANDRhoRhoTEqualsIdEX       ++
 priDTeX "productFormulaeInTeXSHORT"      productFormulaeInTeXSHORT       ++
 priDTeX "productFormulaeInTeXLONG"       productFormulaeInTeXLONG        ++
 printTT "translatedProdFormulaeFstOrd"   translatedProdFormulaeFstOrd    ++
 prinTeX "transltdProdFormulaeTeXSHORT"   transltdProdFormulaeTeXSHORT    ++
 prinTeX "transltdProdFormulaeTeXLONG"    transltdProdFormulaeTeXLONG     ++
 printTS "stripProductFormulae"           stripProductFormulae            ++
 printTS "stripProductFormulaeFstOrd"     stripProductFormulaeFstOrd      ++
 prinMat "piMatEX"                        piMatEX                         ++
 prinMat "piPiTMatEX"                     piPiTMatEX                      ++
 prinMat "rhoRhoTMatEX"                   rhoRhoTMatEX                    ++
 prinMat "rhoMatEX"                       rhoMatEX                          )
           -- Ende ProdTestActualResults
putStr ">SHUTTLE:RelaHaRes:TestOrdner:TestResultMacros< ausgegeben!\n"
```

# 6 Outlook

Over the years there has been a considerable interest of the author to be able to use relations as boolean matrices in the same way as real or complex matrices are used on the computer. This lead my group to initiate several studies as student work, diploma theses, or as byproducts of doctoral theses. During the last two years, when I was member of the TARSKI group (the European COST Action 274: *Theory and Applications of Relational Structures as Knowledge Instruments*) my impression that such techniques should be developed grew even further. I learned that in many application fields — as well as distributed over many locations — considerable but still incoherent work was in progress.

It is this situation which is addressed by this proposal. The relational language is intended to be some sort of a reference language. Colleagues are expressly invited to contribute to it. The proposal is still open for discussion, not least the notation chosen here. The proposal is still incomplete as some cases are not yet programmed and some parts may still be erroneous. The multitude of case decompositions is far from having been tested thoroughly. On the other hand, the HASKELL side of this literate program is heavily used in a diversity of environments — at least by the author. So it will gradually improve.

Several future developments are conceivable, some of which have already been studied to a certain extent. First, a paper on a Rasiowa-Sikorski style proof system for relational theories is close to being finished. It will use the language developed here. Secondly, it should be studied whether it is a good idea to bind the language together with the well-known Isabelle system to have even superior possibilities in theorem proving. Thirdly, there will be some student paper to re““er the former RALF system according to these new standards. As a fourth point, we aim at triggering the RELVIEW machine out of this language using its KURE interface. As a fifth point connection to the RATH system will be made so as to be able to interpret the language in completely different models such as interval algebras, compass algebras, to mention just a few.

## Acknowledgments

This report has been started in April 2003 during a stay in Warsaw visiting Ewa Orłowska at the Institute for Telecomunications. It owes much to the discussions held there. The report is, of course also influenced from all the friends and colleagues who over the years contributed to the RelMiCS initiative (Relational Methods in Computer Science), to which it shall further contribute. Many thanks go first to Michael Ebert, and also to Wolfram Kahl, Eric Offermann, Michael Winter, and numerous others.

# Bibliography


[Apt90] Krzysztof R. Apt. Logic Programming, Chapter 10. In *Handbook of Theoretical Computer Science, Vol. B*, pages 493–574. Elsevier, 1990.

[BBH+99] Ralf Behnke, Rudolf Berghammer, Thorsten Hoffmann, Barbara Leoniuk, and Peter Schneider. Applications of the RelView System. In Rudolf Berghammer and Yassine Lakhnech, editors, *Tool Support for System Specification, Development, and Verification*, Advances in Computing Science, pages 33–47. Springer Vienna, 1999.

[BBMS98] Ralf Behnke, Rudolf Berghammer, E. Meyer, and Peter Schneider. RelView — A System for Calculation With Relations and Relational Programming. In Egidio Astesiano, editor, *Proc. 1st Conf. Fundamental Approaches to Software Engineering*, number 1382 in Lect. Notes in Comput. Sci., pages 318–321. Springer-Verlag, 1998.

[BH01] Rudolf Berghammer and Thorsten Hoffmann. Modeling Sequences within the RelView System. *J. Universal Comput. Sci.*, 7:107–123, 2001.

[BHLM03] Rudolf Berghammer, Thorsten Hoffmann, Barbara Leoniuk, and Ulf Milanese. Prototyping and Programming With Relations. *Electronic Notes in Theoretical Computer Science*, 44(3):24 pages, 2003.

[BSW03] Rudolf Berghammer, Gunther Schmidt, and Michael Winter. RelView and Rath — Two Systems for Dealing with Relations. In de Swart et al. [dSOSR03], pages 1–16. 273 pages.

[BvKU96] Rudolf Berghammer, Burkhard von Karger, and C. Ulke. Relation-Algebraic Analysis of Petri Nets with RelView. In Steffen B. Margaria T., editor, *Proc. 2nd Workshop Tools and Applications for the Construction and Analysis of Systems*, number 1055 in Lect. Notes in Comput. Sci., pages 49–69. Springer-Verlag, 1996.

[dSOSR03] Harrie de Swart, Ewa S. Orłowska, Gunther Schmidt, and Marc Roubens, editors. *Theory and Applications of Relational Structures as Knowledge Instruments.* COST *Action 274:* TARSKI. *ISBN 3-540-20780-5*, number 2929 in Lect. Notes in Comput. Sci. Springer-Verlag, 2003. 273 pages.

[Hat97] Claudia Hattensperger. *Rechnergestütztes Beweisen in heterogenen Relationenalgebren.* PhD thesis, Fakultät für Informatik, Universität der Bundeswehr München, 1997. Dissertationsverlag NG Kopierladen, München, ISBN 3-928536-99-0.

[HBS94] Claudia Hattensperger, Rudolf Berghammer, and Gunther Schmidt. Ralf — A relation-algebraic formula manipulation system and proof checker (Notes to a system demonstration). In Nivat et al. [NRRS94], pages 405–406. Proc. 3rd Int'l

Conf. Algebraic Methodology and Software Technology (AMAST '93), University of Twente, Enschede, The Netherlands, Jun 21–25, 1993. Only included for references from [HBS94].

[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/`.

[KS00] Wolfram Kahl and Gunther Schmidt. Exploring (Finite) Relation Algebras With Tools Written in Haskell. Technical Report 2000/02, Fakultät für Informatik, Universität der Bundeswehr München, October 2000. `http://ist.unibw-muenchen.de/Publications/TR/2000-02/`.

[NRRS94] Maurice Nivat, Charles Rattray, Teodore Rus, and Giuseppe Scollo, editors. *Algebraic Methodology and Software Technology*, Workshops in Computing. Springer-Verlag, 1994. Proc. 3rd Int'l Conf. Algebraic Methodology and Software Technology (AMAST '93), University of Twente, Enschede, The Netherlands, Jun 21–25, 1993. Only included for references from [HBS94].

[OS04a] Ewa S. Orłowska and Gunther Schmidt. Mechanisation of spatial reasoning, 2004. In preparation.

[OS04b] Ewa S. Orłowska and Gunther Schmidt. Rasiowa-Sikorski Proof Systems in Relation Algebra. Technical Report 2004-??, Fakultät für Informatik, Universität der Bundeswehr München, 2004.

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

[Sch03a] Gunther Schmidt. Relational Language. Technical Report 2003-05, Fakultät für Informatik, Universität der Bundeswehr München, 2003. 101 pages, `http://ist.unibw-muenchen.de/Inst2/People/schmidt/RelLangHomePage.html`.

[Sch03b] Gunther Schmidt. Theory Extraction in Relational Data Analysis. In de Swart et al. [dSOSR03], pages 68–86. 273 pages.

[Sch04a] Gunther Schmidt. A Proposal for a Multilevel Relational Reference Language, 2004. 24 pages, to appear in Electronic J. on Relational Methods in Comput. Sci.

[Sch04b] Gunther Schmidt. Implication Structures, 2004. 11 pages, submitted to RelMiCS 8.

[Sch04c] Gunther Schmidt. Partiality I: Embedding Relation Algebras. *Special Issue of the Journal of Logic and Algebraic Programming; edited by Bernhard Möller*, 2004. 29 pages, submitted.

[Sch04d] Gunther Schmidt. Relational Data Analysis. In Rudolf Berghammer, Bernhard Möller, and Georg Struth, editors, *RelMiCS '7 — Relational and Kleene-Algebraic Methods in Computer Science. Proc. of the Internat. Workshop RelMiCS '7 and*

*2nd Internat. Workshop on Applications of Kleene Algebra, in combination with a workshop of the COST Action 274: TARSKI. Revised Selected Papers*, number 3051 in Lect. Notes in Comput. Sci., pages 227–237. Springer-Verlag, 2004. ISBN 3-540-22145-X, 279 pages.

[Win03] Michael Winter. Generating Processes from Specifications Using the Relation Manipulation System RelView. *Electronic Notes in Theoretical Computer Science*, 44(3):27 pages, 2003.