Journal on Relational Methods in Computer Science, Vol. 1, 2004, pp. 314 - 338

A Proposal for a Multilevel Relational Reference Language

Gunther Schmidt*

Institute for Software Technology, Department of Computing Science
Federal Armed Forces University Munich, 85577 Neubiberg
e-Mail: Schmidt@Informatik.UniBw-Muenchen.DE

Abstract. A highly expressive multilevel relational reference language is proposed that covers most possibilities to use relations in practical applications. The language is designed to describe work in a heterogeneous setting. It originated from a Haskell-based system announced in [29], forerunners of which were [17, 16].
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 providing 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.

1 Introduction

When an engineer is about to design an artefact and has to apply Linear Algebra methods (such as solving systems of linear equations or determining eigenvalues and eigenvectors), he will approach the respective computing center and most certainly get the necessary software. When the matrices considered become boolean matrices, i.e., relations, the situation changes dramatically. Neither will one find persons competent in that, nor will there exist commonly accepted high-quality software. Even formulation of the ideas is often bound to the respective scientists personal habits of denotation.

A commonly accepted language that covers at least the broad majority of the topics handled with relational means is not yet available. It is this situation

Received by the editors March 21, 2004, and, in revised form, September 21, 2004.
Published on December 10, 2004.

* Cooperation and communication around this research was partly sponsored by the European Cost Action 274: Tarski (Theory and Applications of Relational Structures as Knowledge Instruments), which is gratefully acknowledged.

which is addressed by the present article. As far as relational research is reported on games, satisfiability, domain construction, e.g., this is not new — new is the exposition of how to formulate all this so as to separate it from its interpretation. 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.

Recollecting [17, 16, 20, 29, 8, 31], a multilevel relational reference language should 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-control 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, thus dealing very efficiently with relations of considerable size [11, 5, 4, 6, 7, 38]. In a third variant, interpretation shall be possible using the Rath-system, a Haskell-based tool 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 [16]. Already now, however, a variant has been initiated that allows proofs in Rasiowa-Sikorski style [22].
– 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. Opposed to these *external* translations, also *internal* ones shall be supported, namely those translating a relational formula in 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, and thus handle the strict/noncontinuous as well as the non-strict/continuous case in a common framework. This means in particular to concentrate on the language used and to scrupulously distinguish which operation to apply.

With regard to all these aspects several studies have shown considerable progress, not least concerning system control. All this cannot be presented in one single article. For the background, we refer to the underlying reports [20, 30].

These underlying reports are written in literate style; they are thus not just scientific texts but also programs and have been used to thoroughly test many of the concepts presented here. For this, we have used HASKELL [18] as the programming language. HASKELL is by far the language best suited for such structural and transformational experiments. It is purely functional and now widely accepted in research and university teaching. For more information about HASKELL see the HASKELL WWW site at

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

For solely studying the language to be developed here, we might have chosen to present it in some grammar. We have, however, supported our study by many programs to investigate the interdependencies and to check our decisions from various points of view with programs in HASKELL. As notation in HASKELL rather closely resembles the grammar structures, we decided to use it also for presentation purposes.

We are fully aware that many people may not be versed enough in HASKELL. So 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 is desired.

The article 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 collection of syntactic material etc. Finally, theories are introduced as HASKELL data structures. Typing, well-formedness, and the most general types are studied in Ch. 3. Several ways of translation of terms, not least to TEX are presented in Ch. 4. Chapter 5 contains the definition of models as HASKELL data structures, followed by all the functions necessary for interpretation in such a model. Chapters 6 and 7 contain various case studies of using the language: Generic constructions, Rasiowa-Sikorski rules. The report ends with an outlook and some acknowledgments.

2 The Multilevel Language in Haskell

A multilevel relational reference 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. 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 The Heterogeneous Setting

From the very beginning, we work in a typed or heterogeneous setting. We admit direct products, sums, and powers to be formed generically. Such typing means that we have to provide for a language to formulate basics of a category. This may seem a difficult step to start with; it is, however, outweighed by a big advantage: Finite models are not excluded as they would be in the homogeneous case.

What one should bear in mind when reading the following data type definitions is that capital first letters such as in `CstO, Elem, Rela` indicate socalled constructors and that the respective data may be matched one against the other. In the case of infix-notated operators, the corresponding is indicated with encapsulation in colons, as in "`:***:`".

```
data CatObjCst = CstO String
data CatObjVar = VarO String  | IndexedVarO String Int
data CatObject = OC CatObjCst | OV CatObjVar | DirPow CatObject |
                 DirPro CatObject CatObject  | QuotMod RelaTerm |
                 DirSum CatObject CatObject  | InjFrom VectTerm | UnitOb
```

Normally, we will be able to give names to the category objects. When formulating proof rules, we will also need variables for category objects. Here and in later cases, we provide for two forms of variable denotations. The first is just a name while the indexed variable name offers more easily an ever expanding set of variable names. The categorical standard constructions of forming the direct product, direct sum, direct power, as well as the unit object are provided for. In addition, we generate dependent types when a "subset" is given of when a quotient is formed. They will require to obey typing discipline.

2.2 Constants and Variables

When working in first-order predicate logic, one will usually need denotations for individual variables and constants. In the present multilevel setting, this applies to all three levels. Therefore, element constants and element variables as well as 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. We restrict ourselves to unary predicates represented by vectors and binary predicates, i.e., relations.

This will, of course, lead to difficult borderline situations: We are at the same time working in a first-order predicate logic for the elements and will via vectors and relations, with the possibility to quantify over these, open the door to second-order logic a tiny bit.

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

data ElemVari  = VarE String CatObject | IndexedVarE String Int CatObject
data VectVari  = VarV String CatObject | IndexedVarV String Int CatObject
data RelaVari  = VarR String CatObject CatObject |
                 IndexedVarR String Int CatObject CatObject
```

The function constant may not really be necessary as we have relation constants. A relational constant is nothing else 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. Again, the possibility of defining indexed variable names is given.

2.3 Terms

All this allows to build first-order predicate logic introducing terms and formulae on either one of the three levels. According to our notation, vectors are best conceived as column vectors. From the beginning, we distinguish element terms, vector terms, and relation terms. Null, universal, and identity relation constants will be given generically. The generic transitions between the three levels considered will later need care. `VectToElem`, e.g., provides the transition from a vector to the corresponding element in the powerset, while `RelaToVect` converts a relation to the corresponding vector in the direct product.

```
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
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
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 |
RFctAppl RelaFct RelaTerm
```

Constructs such as `:||||:`, `:|||:`, `:&&&&:`, `:&&&:`, `:***:`, `Convs`, `NegaV`, `NegaR` don't need detailed explanation; they resemble union, intersection, composition, conversion, and negation of vector and relational terms, resp. The element terms constructed via `SomeV`, `SomeR`, `ThatV`, `ThatR`, however, deserve explanation. They are correctly defined only if, e.g., the vector term `vt` in `ThatV vt` denotes a point. In `SomeR rt`, the relational term `rt` must denote a nonempty part of the identity. Later, typically a proof obligation will be issued to guarantee such properties.

Further transitions lead from the element level to the others by `PointVect`, `PointDiag`. Given an element term one may generate the corresponding "singleton set" vector or diagonal relation "with just one single element" in the diagonal. The function applications `EFctAppl`, `VFctAppl`, `RFctAppl` refer to the function definitions to be defined in the next subsection.

The construct `RelaTerm :****: VectTerm` is intended to model the Peirce product. (In our favourite model relations are always boolean matrices `[[Bool]]`, while vectors are lists `[Bool]` as opposed to one-column matrices. So we need a different symbol for the mixed product.) With `Pi`, `Rho`, `Pair`, generic denotations for projections from a direct product are introduced; in the same way `Iota`, `Kappa`, `CASE` provide generic denotations for the injections into a direct sum. For these generic constructs see our later Sect. 6. Finally, `Epsi` generically denotes the relationship between a set and its powerset. A (column) vector multiplied via `:||--:` with a (row) vector will deliver a relation. `Project` converts an equivalence to the mapping onto the quotient.

The operations `:\/:` and `:*:` are defined using the other operations. The first resembles the often discussed fork operator ∇, while the second expresses the corresponding parallel propagation. Also the two symmetric quotients `Syq`, `SyQ` are defined using other operations. As we have included them here, one may later match them. When using such defined constructs, an expansion will always take place as a first step via `expandDefine` ...

2.4 Functions

The following are necessary when, e.g., introducing a transitive closure of a relation by the classical infimum definition. Here it may be discussed whether also variables for such functions should be introduced.

```
data ElemFct  = EFCT ElemVari ElemTerm
data VectFct  = VFCT VectVari VectTerm
data RelaFct  = RFCT RelaVari RelaTerm
```

Often a relation is given descriptively, e.g., saying that it is the least fixedpoint of some functional. While it is usually not a good idea to use a purely descriptive definition to compute the relation, it may well be the starting point for proving that a given algorithm really works. Using the facility just introduced, it is possible to define the two example functionals for transitive and difunctional closure. The function `supply` gives an appropriate number of indexed variables introducing the necessary category object variables with a sufficiently high index starting (here) from 99.

```
transFctl, difuFctl :: RelaTerm -> RelaFct
transFctl r =
   let ([],[],[],[rv],[]) = supply 99 0 0 0 1 0
       rt = RV rv
   in  generalTypeOfRelaFct $ RFCT rv (r :|||: (rt :***: rt))
difuFctl r =
   let ([],[],[],[rv],[]) = supply 99 0 0 0 1 0
       rt = RV rv
   in  generalTypeOfRelaFct $
          RFCT rv (r :|||: (rt :***: (Convs rt) :***: rt))
```

When instantiated with R for `r`, these are translated into TEX as

$\langle\backslash X \longrightarrow R \cup X;X\rangle$ $\quad\langle\backslash X_{\mathrm{O}_1,\mathrm{O}_1} \longrightarrow R_{\mathrm{O}_1,\mathrm{O}_1} \cup X_{\mathrm{O}_1,\mathrm{O}_1};X_{\mathrm{O}_1,\mathrm{O}_1}\rangle$

$\langle\backslash X \longrightarrow R \cup X;X^{\mathsf{T}};X\rangle$ $\quad\langle\backslash X_{\mathrm{O}_1,\mathrm{O}_2} \longrightarrow R_{\mathrm{O}_1,\mathrm{O}_2} \cup X_{\mathrm{O}_1,\mathrm{O}_2};X_{\mathrm{O}_1,\mathrm{O}_2}{}^{\mathsf{T}};X_{\mathrm{O}_1,\mathrm{O}_2}\rangle$

2.5 Sets of Elements, Vectors, and Relations

In order to be able to write down formulae on least upper bounds, e.g., also sets of elements, vectors, and relations shall be formed. They are provided in one of two possible forms.

```
data ElemSET = VarES String CatObject |
               ES ElemVari [Formula]  | EX [ElemTerm] CatObject

data VectSET = VarVS String CatObject |
               VS VectVari [Formula]  | VX [VectTerm] CatObject
```

```
data RelaSET = VarRS String CatObject CatObject |
               RS RelaVari [Formula]  | RX [RelaTerm] CatObject CatObject
```

Either sets are given by some condition or as an explicit set. For the explicit sets also the type is provided, a measure which is relevant only in case the set is void, i.e., in constructs such as `RX [] O1 O2`.

Using the relation set facility, one may formulate the least fixedpoint operation.

```
leastFixedPoint fctl =
   let ([],[],[],[rv],[]) = supply 999 0 0 0 1 0
       rt = RV rv
       relaSet = RS rv [RF $ RFctAppl fctl rt :<==: rt]
   in  InfRela relaSet
```

Here the functional is a parameter that we now instantiate in two ways in order to obtain two definitions for the transitive as well as for the difunctional closure.

```
transClosure = leastFixedPoint transFctl
difuClosure  = leastFixedPoint difuFctl
```

It had obviously been necessary to use formulae which we introduce next.

2.6 Formulae

Four sorts of formulae are distinguished in order to maintain type control as long as possible. Only when negation, e.g., is applied to a formula `f = Disjunct g h`, it will be handled as a formula: `Negated f`. Until that point, i.e. as long as negation is something like $A \not\subseteq B$, the type is a convenient way of correctness control. In addition, it allows pattern matching.

```
data UnivOrExist = Univ | Exis
data ElemForm =
   Equation ElemTerm  ElemTerm | NegaEqua ElemTerm ElemTerm |
   QuantElemForm UnivOrExist ElemVari [Formula]
data VectForm =
   VectTerm :<===: VectTerm | VectTerm :>===: VectTerm |
   VectTerm :====: VectTerm | VectTerm :<=/=: VectTerm |
   VectTerm :==/=: VectTerm | VectTerm :>=/=: VectTerm |
   VE VectTerm ElemTerm | VectInSet VectTerm VectSET |
   QuantVectForm UnivOrExist VectVari [Formula]
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]
```

Element terms may just be equal or unequal, while vector or relation terms may in addition be compared with regard to containment.

The basic multilevel connection shows up in `VE vt et` meaning $et \in vt$, i.e., that the element designated by the element term `et` is contained in the vector designated by the vector term `vt`, and `REE rt et1 et2` meaning $(et1, et2) \in rt$, or that the element pair `(et1, et2)` is in relation `rt`. Quantification over vectors as unary predicates and relations as binary predicates moderately opens the door to second-order predicate logic.

The result type is in all cases intended to be `Bool`; we have, however, tried to benefit from typing of vectors and relations as long as possible. Only now, we bind these three variants of formulae together as follows.

```
data FormVari = VarF String | IndexedVarF String Int
data Formula  = FV FormVari | EF ElemForm | VF VectForm | RF RelaForm |
                Verum | Falsum | Negated Formula |
                Implies  Formula Formula | SemEqu   Formula Formula |
                Disjunct Formula Formula | Conjunct Formula Formula
```

2.7 Theories

We are now in a position to formulate theory presentations.

```
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
```

We have decided to not include an `ElemFct` as these may easily be simulated by relations. There exist several auxiliary functions to test whether a theory is formulated in a correct way. Later one may check whether some proposed model is indeed a model of the theory.

Over these definitions the usual recursive algorithms are defined. The syntactical material may be collected with `syntMatUsed`, accumulating them as a tuple (category object variables, category object constants, element variables, element constants, vector variables, vector constants, relation variables, and relation constants). Free and bound variables may, of course, also be determined.

3 Typing

Every term is supposed to have category objects assigned for typing purposes. Such a type may be given explicitly. It may, however, also be deduced from the construction of the term in question. So we will often obtain types by reasoning.

3.1 Typing Discipline

As we intend to define a language supporting work with polymorphically typed heterogeneous relations, we have to provide for such reasoning about typing. A corresponding type inference system has already been proposed in [19], mainly based on [3, 1].

We start determining domain and range of a construct. Collecting this in a type class definition, one may henceforth simply write `dom,cod,typeOf`. The typical checks are provided for well-formedness using `isWellFormed`.

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

Collecting type restrictions starts from, e.g., $A;B$, from which we infer that `cod A = dom B`. When a set of terms and/or formulae is given, we first collect all such type restrictions. When comparing category objects in this way, one may find out that they cannot be made equal, in which case `Nothing` is returned. If they are equal, `Just []` is returned. In other cases, `Just` is returned with a list of category object pairs that need to be unified to make them equal.

3.2 Most General Typing

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 unification algorithm we apply is an adaptation and implementation of the article of Krzysztof Apt in the Handbook of Theoretical Computer Science, vol. B, [2], so it does not need additional comments. 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.

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 (set of) terms, or (set of) 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
```

4 Translation of Formula

Once terms and/or formulae are built and well-formed, one will immediately start transforming them in one way or the other so as to achieve certain goals. A main example is transformation within some proof system. But also transformation to TEX-text or Ascii-text is some sort of a translation.

4.1 Translation into TEX

To make the type-carrying language proposed here more readable, we provide for a translation into TEX. In order to facilitate all this we have defined a type class

```
class TeX a where
   tEX :: Bool -> a -> String
```

with `tEX` allowing to transfer constructs automatically to TEX-notation. All examples in this article have been generated in this way. The boolean switch is available in order to switch from a long and detailed form to a shorter one without typing information.

4.2 Translation to First-Order Form

If in an environment first-order formulae are supposed to be provided, one often feels that it has been rather cumbersome and error-prone to write them down. In such cases, one will often formulate in a higher relational language and afterwards translate to first-order form — again a translation.

Translation of relational or vector formulae to the element form means in particular to introduce all the individual variables necessary as well as quantifications which are hidden in the more complex relational form, e.g.,

$A \subseteq B$ as opposed to $\forall x, y : (x, y) \in A \rightarrow (x, y) \in B$

In our multilevel approach both are legitimate forms. In some sense one will say that both forms express the same. However, this is true only for representable relation algebras. But there exist also non-representable ones. Translation from one form to the other is possible and is included in the language definition. As we have to generate the variable names x, y in the course of the translation, we should take care, that they don't interfere with already existing ones. We have, therefore, introduced some accounting on the variables already used.

For the translation between the levels, there exists a difficult borderline separating first-order logic and relational logic in the form explained here. When quantifying over 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, the relational language is burdened with the existence of non-standard models.

5 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. Here a difference arises between the element layer on one side and the vector and relational layer on the other. While the element layer may be interpreted in just one way, the relational and the vector layer sometimes admit two.

Relation algebras may be non-representable ones. These can often be interpreted using the Rath system. To this end one had to program code bridging the gap between the two systems, what has not yet been done.

For a representable relation algebra it is in addition possible, to use the translation into first-order formulation and then interprete this resulting in matrices conceived as binary predicates. Two forms of such an interpretation are possible, from which the first will later work via emitting a string with which the Relview system may be triggered. The second uses the following standard mechanisms. It will, however, not be possible to interprete a non-representable relation algebra in this standard way.

5.1 The Standard Model

Our standard model is available for a representable relation algebra. 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
data InterpretCons = InterCon ElemConst   Int
data InterpretVect = InterVec VectConst  [Bool]
data InterpretRela = InterRel RelaConst [[Bool]]
data InterpretFunc = InterFct FuncConst  [Int]
data InterpretVFct = InterVFc VectFct   ([Bool]  ->  [Bool])
data InterpretRFct = InterRFc RelaFct  ([[Bool]] -> [[Bool]])
```

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 = RATHModel    String   |
             RELVIEWModel String   |
             MO String            -- name 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
```

The first two variants are just indications where to embed possible future models extending the present ideas. 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. Lots of technicalities are necessary to ensure that this works as it is supposed to, but we do not explain this here.

5.2 Interpretation in the Standard Model

Smaller problems should be investigated without crossing the borderline to other systems such as RelView. In these cases, the following interpretation may be taken. Before the interpretation is possible, we need valuations of the individual variables.

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

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

Now we can start interpreting items of the language. We show types of these functions only, omitting the function bodies which may be found in the report.

```
interpretElemConst :: Model -> ElemConst ->   Int
interpretVectConst :: Model -> VectConst ->  [Bool]
interpretRelaConst :: Model -> RelaConst -> [[Bool]]
```

Based on these interpretations of constants, interpretation proceeds as on would expect.

```
interpretElemTerm :: Model -> Env -> ElemTerm ->     Int
interpretVectTerm :: Model -> Env -> VectTerm ->    [Bool]
interpretRelaTerm :: Model -> Env -> RelaTerm ->   [[Bool]]
interpretVectFct  :: Model -> Env -> VectFct  ->    [Bool]  ->  [Bool]
interpretRelaFct  :: Model -> Env -> RelaFct  ->   [[Bool]] -> [[Bool]]
interpretVectSET  :: Model -> Env -> VectSET  ->   [[Bool]]
interpretRelaSET  :: Model -> Env -> RelaSET  -> [ [[Bool]] ]
interpretElemForm :: Model -> Env -> ElemForm ->     Bool
interpretVectForm :: Model -> Env -> VectForm ->     Bool
interpretRelaForm :: Model -> Env -> RelaForm ->     Bool
interpretFormula  :: Model -> Env -> Formula  ->     Bool
```

One may have observed that generic constructs such as `Pi, Rho, Iota, Kappa, Epsi` have not been mentioned here. Interpretations for these are again generated automatically by the system as shown by the following example.

```
obj1 = OC $ CstO "Obj1"            -- definition of 2 category objects
obj2 = OC $ CstO "Obj2"
rel1 = Rela "R" obj1 obj1          -- definition of 2 relation constants
rel2 = Rela "S" obj2 obj2
mat1 = [[True, False,True ],       -- interpreting boolean matrices
        [False,False,False],
        [True, False,True ]]
mat2 = [[True, False,False,True ],
        [False,True, True, False],
        [False,True, True, False],
        [True, False,False,True ]]
thTEST = TH "Sparse Test Theory"    -- theory definition
            [obj1, obj2] [] []
            [rel1, rel2] [] [] [] []
moTEST = MO "Simple Test Model"     -- model definition
            [Carrier obj1 3, Carrier obj2 4] [] []
            [InterRel rel1 mat1, InterRel rel2 mat2] [] [] []
```

One may now proceed and define π = `Pi obj1 obj2`, ρ = `Rho obj1 obj2` as well as $R{:}{*}{:}S$ = `(RC rel1) :*: (RC rel2)`. With

`interpretRelaTerm moTEST ([],[],[])` ...

all three may be interpreted in an "empty" environment giving as TEX-output

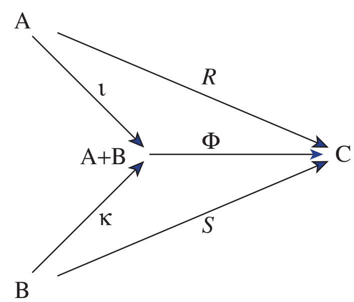
$$\pi = \begin{pmatrix} 1&0&0\\ 1&0&0\\ 1&0&0\\ 1&0&0\\ 0&1&0\\ 0&1&0\\ 0&1&0\\ 0&1&0\\ 0&0&1\\ 0&0&1\\ 0&0&1\\ 0&0&1 \end{pmatrix} \quad R:*:S = \begin{pmatrix} 1&0&0&1&0&0&0&0&1&0&0&1\\ 0&1&1&0&0&0&0&0&0&1&1&0\\ 0&1&1&0&0&0&0&0&0&1&1&0\\ 1&0&0&1&0&0&0&0&1&0&0&1\\ 0&0&0&0&0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&0&0&0&0&0\\ 1&0&0&1&0&0&0&0&1&0&0&1\\ 0&1&1&0&0&0&0&0&0&1&1&0\\ 0&1&1&0&0&0&0&0&0&1&1&0\\ 1&0&0&1&0&0&0&0&1&0&0&1 \end{pmatrix} \quad \rho = \begin{pmatrix} 1&0&0&0\\ 0&1&0&0\\ 0&0&1&0\\ 0&0&0&1\\ 1&0&0&0\\ 0&1&0&0\\ 0&0&1&0\\ 0&0&0&1\\ 1&0&0&0\\ 0&1&0&0\\ 0&0&1&0\\ 0&0&0&1 \end{pmatrix}$$

6 Generic Constructs as a Running Example

As a running example exhibiting the usefulness of the language proposed, we use the following generic constructs. This will also clarify them. With these generic constructions, we in addition demonstrate transition to the TEX-form. In all three cases we start with a mathematical explanation in TEX-form. This form, however, is the result of applying `tEX` to the characterizing formulae formulated in the language proposed here.

6.1 Characterisation of Direct Sums

The direct sum in its simplest form resembles a disjoint union of two sets. When in addition some algebraic structure is present, by mathematical folklore a "universal characterisation" is given saying that the sum structure is *uniquely characterized up to isomorphism*. Such a universal characterisation ranges over all sets C carrying the structure in question and all mappings R, S. Some sort of a preordering of (C, R, S) via the possibility of factorising is introduced and the definition asserts that some sort of an infimum $(A + B, \iota, \kappa)$ will be obtained.



Universal characterisation of the direct sum

This method is, thus, purely descriptional. Even if a sum candidate is presented, it cannot be tested along this definition: Quantification runs *over all sets* carrying the structure and *over all mappings* leading to A, B; the characterisation is not even first-order. So it is important that, when working with heterogeneous relations, one may also give an *equational* definition instead.

Over a long period of time, relation algebraists were accustomed to work homogeneously; see not least [36]. This made concepts difficult, as the well-established typing mechanisms a computer scientist applies routinely had to be replaced developing ad hoc mathematics.

It seems that homomorphisms of heterogeneous structures (graphs, programs, e.g.) have first been formalised relationally during the Winter term 1974/75 at *Technische Universität München* in the lectures on *Graphentheorie* by Gunther Schmidt. The notes [32] of these have been printed as an internal report of the *Institut für Informatik*. This was then used in [25–28].

Once homomorphisms had been formalised, the characterising formulae for direct sums, direct products, and direct powers were formulated and could further be investigated in diploma theses at the Technische Universität München. Initiated by Gunther Schmidt together with Rudolf Berghammer, such theses were carried out in [35, 13, 39] by Ingrid Taferner, Rodrigo Cardoso, and Hans Zierer.

The first publication of the equational characterisations seems to have been presented with the series of publications [9, 37] and [24, 33, 10, 34], and not least [12, 40, 41, 15], which followed the diploma theses mentioned.

The sum-characterising formulae — in a short and in a long version with type information — are as follows.

$$\begin{array}{ll}
\iota;\iota^{\mathsf{T}} = \mathbb{I}, & \iota_{O_1,O_1+O_2};\iota^{\mathsf{T}}_{O_1,O_1+O_2} = \mathbb{I}_{O_1}, \\
\kappa;\kappa^{\mathsf{T}} = \mathbb{I}, & \kappa_{O_2,O_1+O_2};\kappa^{\mathsf{T}}_{O_2,O_1+O_2} = \mathbb{I}_{O_2}, \\
\iota;\kappa^{\mathsf{T}} \subseteq \perp\!\!\!\perp, & \iota_{O_1,O_1+O_2};\kappa^{\mathsf{T}}_{O_2,O_1+O_2} \subseteq \perp\!\!\!\perp_{O_1O_2}, \\
\iota^{\mathsf{T}};\iota \cup \kappa^{\mathsf{T}};\kappa = \mathbb{I} & \iota^{\mathsf{T}}_{O_1,O_1+O_2};\iota_{O_1,O_1+O_2} \cup \kappa^{\mathsf{T}}_{O_2,O_1+O_2};\kappa_{O_2,O_1+O_2} = \mathbb{I}_{O_1+O_2}
\end{array}$$

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

$$\begin{array}{l}
\langle \forall x:\ \langle \forall y:\ \langle \exists u:\ (x,u) \in \iota\ \wedge\ (y,u) \in \iota \rangle\ \Longrightarrow\ x = y \rangle \rangle\ \wedge \\
\qquad\qquad \langle \forall x:\ \langle \exists v:\ (x,v) \in \iota \rangle \rangle, \\[1ex]
\langle \forall x:\ \langle \forall y:\ \langle \exists u:\ (x,u) \in \kappa\ \wedge\ (y,u) \in \kappa \rangle\ \Longrightarrow\ x = y \rangle \rangle\ \wedge \\
\qquad\qquad \langle \forall x:\ \langle \exists v:\ (x,v) \in \kappa \rangle \rangle, \\[1ex]
\langle \forall x:\ \langle \forall y:\ \neg\,(\langle \exists u:\ (x,u) \in \iota\ \wedge\ (y,u) \in \kappa \rangle) \rangle \rangle, \\[1ex]
\langle \forall x:\ \langle \forall y:\ \langle \exists u:\ (u,x) \in \iota\ \wedge\ (u,y) \in \iota \rangle\ \vee \\
\qquad\qquad \langle \exists u:\ (u,x) \in \kappa\ \wedge\ (u,y) \in \kappa \rangle\ \Longrightarrow\ x = y \rangle \rangle\ \wedge \\
\qquad\qquad \langle \forall x:\ \langle \exists v:\ (v,x) \in \iota \rangle\ \vee\ \langle \exists v:\ (v,x) \in \kappa \rangle \rangle
\end{array}$$

In our language an equational universal characterisation may be formulated. Two category objects are bound together using two injective mappings ι, κ satisfying the following formulae

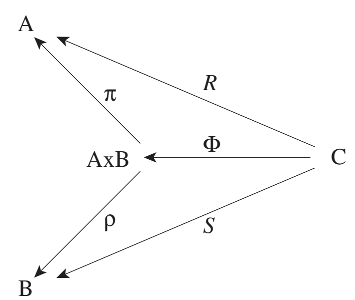
```
sumCharacterizingFormulae o1 o2 =
   let iota   = Iota  o1 o2
       kappa  = Kappa o1 o2
       iotaT  = Convs iota
       kappaT = Convs kappa
       iiT = iota   :***: iotaT
       kkT = kappa  :***: kappaT
       iTi = iotaT  :***: iota
```

```
      kTk = kappaT :***: kappa
      ikT = iota   :***: kappaT
  in  [RF $ iiT :===: Ident o1,
       RF $ kkT :===: Ident o2,
       RF $ ikT :<==: NullR o1 o2,
       RF $ iTi :|||: kTk :===: Ident (DirSum o1 o2)]

sumTheory o1 o2 = TH "Sum-Theory" [o1,o2] [] [] [] [] [] []
                     (sumCharacterizingFormulae o1 o2)
```

6.2 Characterization of Direct Products

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



Universal characterisation of the direct product

$$\begin{array}{ll}
\pi^{\mathsf{T}};\pi = \mathbb{I}, & \pi^{\mathsf{T}}_{O_1\times O_2,O_1};\pi_{O_1\times O_2,O_1} = \mathbb{I}_{O_1}, \\
\rho^{\mathsf{T}};\rho = \mathbb{I}, & \rho^{\mathsf{T}}_{O_1\times O_2,O_2};\rho_{O_1\times O_2,O_2} = \mathbb{I}_{O_2}, \\
\mathbb{T} \subseteq \pi^{\mathsf{T}};\rho, & \mathbb{T}_{O_1 O_2} \subseteq \pi^{\mathsf{T}}_{O_1\times O_2,O_1};\rho_{O_1\times O_2,O_2}, \\
\pi;\pi^{\mathsf{T}} \cap \rho;\rho^{\mathsf{T}} = \mathbb{I} & \pi_{O_1\times O_2,O_1};\pi^{\mathsf{T}}_{O_1\times O_2,O_1} \cap \rho_{O_1\times O_2,O_2};\rho^{\mathsf{T}}_{O_1\times O_2,O_2} = \mathbb{I}_{O_1\times O_2}
\end{array}$$

The same formulae are now first translated to first-order form and then automatically to TEX.

$$\langle \forall x: \ \langle \forall y: \ \langle \exists u: \ (u,x) \in \pi \ \wedge \ (u,y) \in \pi \rangle \implies x = y \rangle\rangle \ \wedge \\ \langle \forall x: \ \langle \exists v: \ (v,x) \in \pi \rangle\rangle,$$

$$\langle \forall x: \ \langle \forall y: \ \langle \exists u: \ (u,x) \in \rho \ \wedge \ (u,y) \in \rho \rangle \implies x = y \rangle\rangle \ \wedge \\ \langle \forall x: \ \langle \exists v: \ (v,x) \in \rho \rangle\rangle,$$

$$\langle \forall x: \ \langle \forall y: \ \langle \exists u: \ (u,x) \in \pi \ \wedge \ (u,y) \in \rho \rangle\rangle\rangle,$$

$$\langle \forall x : \langle \forall y : \langle \exists u : (x,u) \in \pi \wedge (y,u) \in \pi \rangle \wedge \\ \langle \exists u : (x,u) \in \rho \wedge (y,u) \in \rho \rangle \implies x = y \rangle \rangle \wedge \\ \langle \forall x : \langle \exists v : (x,v) \in \pi \rangle \wedge \langle \exists v : (x,v) \in \rho \rangle \rangle$$

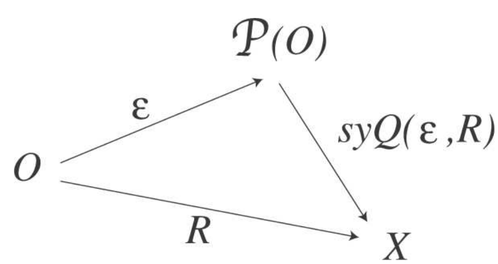
```
prodCharacterizingFormulae o1 o2 =
   let ppi = Pi  o1 o2
       rho = Rho o1 o2
       ppiT = Convs ppi
       rhoT = Convs rho
       ppiTppi = ppiT :***: ppi
       rhoTrho = rhoT :***: rho
       ppippiT = ppi  :***: ppiT
       rhorhoT = rho  :***: rhoT
   in  [RF $ ppiTppi  :===: Ident o1,
        RF $ rhoTrho  :===: Ident o2,
        RF $ UnivR o1 o2 :<==: (ppiT :***: rho),
        RF $ ppippiT :&&&: rhorhoT :===: Ident (DirPro o1 o2)]
prodTheory o1 o2 = TH "Prod-Theory" [o1,o2] [] [] [] [] [] []
                      (prodCharacterizingFormulae o1 o2)
```

6.3 Characterization of Direct Powers

Yet another universally characterized construct is the direct power. It models the is_element_of relation between a set O and its powerset $\mathcal{P}(O)$. We model this with a relation ϵ satisfying in short resp. long form

$\mathtt{syq}(\epsilon, \epsilon) \subseteq \mathbb{I}$, $\qquad \mathtt{syq}(\epsilon_O, \epsilon_O) \subseteq \mathbb{I}_{\mathcal{P}(O)}$

$\forall v : \langle \mathbb{T} \subseteq \mathbb{T} ; \mathtt{syq}(\epsilon, v) \rangle$, $\qquad \langle \forall v_{\mathrm{O}} \subseteq \mathrm{O} : \mathbb{T}_{1\!\!1} \subseteq \mathbb{T}_{1\!\!1 \mathcal{P}(\mathrm{O})} ; \mathtt{syq}(\epsilon_{\mathrm{O}}, v_{\mathrm{O}}) \rangle$



Universal characterisation of the direct power

```
powerCharacterizingFormulae o1 =
   let epsi = Epsi o1
       vvv  = VarV "v" o1
       al   = UnivR UnitOb (DirPow o1)
       syQEpsiEpsi = SyQ epsi epsi
       syQv vv = Syq epsi vv
   in  [RF (syQEpsiEpsi :<==: (Ident (DirPow o1))),
```

```
        VF (UnivQuantVectForm vvv
              [VF (UnivV UnitOb :<===: (al :****: (syQv (VV vvv))))])]
powerTheory o1 = TH "Power-Theory" [o1] []
                   [] [] [] [] [] (powerCharacterizingFormulae o1)
```

7 Further Applications

Further examples shall demonstrate that quite an area of applications may be covered. To this end we first formulate the Dedekind and Schröder rules. Then hints are given to Rasiowa-Sikorski style proofs.

7.1 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.

First, however, we provide for an automatic object, variable, and constant supply. It is indispensable in order to avoid interference between variables in rules and in the items one is going to apply the rules to. By determining the maximum index used in the item and then putting all the rule variables above that start index, one will avoid such problems.

```
dedekindForm sI =
   let ([],[],[],[pv,qv,rv],_) = supply sI 0 0 0 3 0
       [p,q,r] = map RV [pv,qv,rv]
   in  (p :***: q :&&&: r)  :<==:
        ((p :&&&: (r :***: (Convs q)) :***: (q :&&&: (Convs p :***: r))))
correctDedekindRelaForm = generalTypeOfRelaForm $ dedekindForm 15
```

Printing `dedekindForm 1` without determining the general type first shows that ever new category object variables are taken and the result is not well-formed.

$$A_{o_2,o_5};B_{o_3,o_6} \cap C_{o_4,o_7} \ \subseteq \ (A_{o_2,o_5} \cap C_{o_4,o_7};B_{o_3,o_6}{}^{\mathsf{T}});(B_{o_3,o_6} \cap A_{o_2,o_5}{}^{\mathsf{T}};C_{o_4,o_7})$$

In contrast the `correctDedekindRelaForm` is printed in short as well as in long form as follows:

$$A;B \cap C \ \subseteq \ (A \cap C;B^{\mathsf{T}});(B \cap A^{\mathsf{T}};C)$$
$$A_{O_1,O_2};B_{O_2,O_3} \cap C_{O_1,O_3}$$
$$\subseteq \ (A_{O_1,O_2} \cap C_{O_1,O_3};B_{O_2,O_3}{}^{\mathsf{T}});(B_{O_2,O_3} \cap A_{O_1,O_2}{}^{\mathsf{T}};C_{O_1,O_3})$$

As a corresponding example we now show the Schröder rules.

```
schroederAFormula sI =
   let [a,b,c] = map RV $ take 3 $ aRVS sI
       ab = a :***: b
       aTcBar = Convs a :***: (NegaR c)
   in  generalTypeOfFormula $
          SemEqu (RF (ab :<==: c)) (RF (aTcBar :<==: (NegaR b)))

schroederBFormula sI =
   let [a,b,c] = map RV $take 3 $ aRVS sI
       ab = a :***: b
       cBarbT = NegaR c :***: (Convs b)
   in  generalTypeOfFormula $
          SemEqu (RF (ab :<==: c)) (RF (cBarbT :<==: (NegaR a)))
```

Their TEX-representations with and without typing are

$$A{;}B \subseteq C \longleftrightarrow A^{\mathsf{T}}{;}\overline{C} \subseteq \overline{B}$$
$$A_{O1,O2}{;}B_{O2,O3} \subseteq C_{O1,O3} \longleftrightarrow A_{O1,O2}{}^{\mathsf{T}}{;}\overline{C_{O1,O3}} \subseteq \overline{B_{O2,O3}}$$

$$A{;}B \subseteq C \longleftrightarrow \overline{C}{;}B^{\mathsf{T}} \subseteq \overline{A}$$
$$A_{O1,O2}{;}B_{O2,O3} \subseteq C_{O1,O3} \longleftrightarrow \overline{C_{O1,O3}}{;}B_{O2,O3}{}^{\mathsf{T}} \subseteq \overline{A_{O1,O2}}$$

7.2 Proofs in Rasiowa-Sikorski Style

There is a Polish tradition of proving relational formulae in Rasiowa-Sikorski style. To this end one uses rules such as

$$\cup \quad \frac{x^{P \cup Q}y}{x^{P}y,\ x^{Q}y}$$

$$; \quad \frac{x^{P;Q}y}{x^{P}p,\ x^{P;Q}y \quad | \quad p^{Q}y,\ x^{P;Q}y}$$

where p is an arbitrary variable

which are applied in an expanding direction so as to obtain Rasiowa-Sikorski trees. In these trees one will observe whether all subtrees are "closed". A leaf is closed when it is obviously true since a relational expression together with its negative is available. A non-leaf trees is closed if all its subtrees are.

$$\frac{\dfrac{\mathtt{DC} = \overline{\mathtt{C}}}{\mathtt{DC} \subseteq \overline{\mathtt{C}} \qquad | \qquad \overline{\mathtt{C}} \subseteq \mathtt{DC}}}{x^{\overline{\mathtt{DC}}}y,\ x^{\overline{\mathtt{C}}}y}$$

One will easily recognize that the vertical bar between the subtrees means *and*. The variables x, y which appear when switching from the relational to the elementwise consideration are universally quantified where comma-separation means *or*. Such a Rasiowa-Sikorski proof system heavily needs a language together with a system as proposed here to support rule application as well as to present trees in a comprehendable form.

8 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 in tasks such as solving a linear equation or determining an eigenvalue are used by an engineer. 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 the present proposal. The relational language is intended to be some sort of a reference language. Colleagues are expressly invited to use it, discuss it, and contribute to it. It is still open for discussion, not least regarding the notation chosen here. The proposal is still incomplete as some cases are not yet programmed. 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 reengineer 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.

We anticipate that there may be objections against the language Haskell, which is developing rapidly but not yet commonly accepted. It is on the other hand side the one best suited for the endeavour presented here. To resolve the obvious conflict, we aim at the following procedure. We will try to make the Haskell system a stand-alone application. It will come equipped with a front end enabling a user to introduce his or her own favourite notation — assumed to be reasonably expressive — which then will automatically be translated to the language proposed here and handled using it. For the results a similar retranslation will be available.

Acknowledgments

The article is, of course, influenced by 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, not least from the Cost Action 274 Tarski.

References

1. T. M. S. Abramsky and D. M. Gabbay, editors. *Handbook of Logic in Computer Science*, volume 2. Oxford University Press, 1992.
2. Krzysztof R. Apt. Logic Programming, Chapter 10. In *Handbook of Theoretical Computer Science, Vol. B*, pages 493–574. Elsevier, 1990.
3. Henk P. Barendregt. Lambda calculi with types. In Abramsky and Gabbay [1].
4. 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.
5. 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.
6. Rudolf Berghammer and Thorsten Hoffmann. Modeling Sequences within the RelView System. *J. Universal Comput. Sci.*, 7:107–123, 2001.
7. 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.
8. Rudolf Berghammer, Gunther Schmidt, and Michael Winter. RelView and Rath — Two Systems for Dealing with Relations. In de Swart et al. [14], pages 1–16. 273 pages.
9. Rudolf Berghammer, Gunther Schmidt, and Hans Zierer. Symmetric quotients. Technical Report TUM-INFO 8620, Technische Universität München, Institut für Informatik, 1986.
10. Rudolf Berghammer, Gunther Schmidt, and Hans Zierer. Symmetric quotients and domain constructions. *Information Processing Letters*, 33(3):163–168, 1989/90.
11. 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.

12. Rudolf Berghammer and Hans Zierer. Relational algebraic semantics of deterministic and nondeterministic programs. *Theoret. Comput. Sci.*, 43:123–147, 1986.
13. Rodrigo Cardoso. Untersuchungen von parallelen Programmen mit relationenalgebraischen Methoden, 1982. Diploma thesis at *Technische Universität München* supervised by Gunther Schmidt in co-operation with Rudolf Berghammer.
14. 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.
15. Thomas Gritzner. wp-Kalkül und relationale Spezifikation kommunizierender Systeme. Dissertation, 1996.
16. 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.
17. 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. [21], 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 [17].
18. 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/.
19. Claudia Hattensperger and Peter Kempf. Towards a formal framework for hetereogeneous relation algebra. *Information Sciences*, 119:193–203, 1992.
20. 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/.
21. 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 [17].
22. Ewa S. Orłowska and Gunther Schmidt. Rasiowa-Sikorski Proof Systems in Relation Algebra. Technical Report 2004-??, Fakultät für Informatik, Universität der Bundeswehr München, 2004.
23. Gunther Schmidt, Rudolf Berghammer, and Hans Zierer. Beschreibung semantischer Bereiche mit Keimen. Technical Report TUM-I8611, Institut für Informatik, Technische Universität München, 1986. 33 p.
24. Gunther Schmidt, Rudolf Berghammer, and Hans Zierer. Describing semantic domains with sprouts. In Franz-Josef Brandenburg, G. Vidal-Naquet, and Martin Wirsing, editors, *Proc. 4th Symposium on Theoretical Aspects of Computer Science (STACS '87)*, volume 247 of *Lect. Notes in Comput. Sci.*, pages 299–310. Springer-Verlag, February 1987. Gekürzte Version von [23].
25. Gunther Schmidt. Eine relationenalgebraische Auffassung der Graphentheorie. Technical Report 7619, Fachbereich Mathematik der Technischen Universität München, 1976.
26. Gunther Schmidt. Programme als partielle Graphen. Habil. Thesis 1977 und Bericht 7813, Fachbereich Mathematik der Techn. Univ. München, 1977. English as [27, 28].
27. Gunther Schmidt. Programs as partial graphs I: Flow equivalence and correctness. *Theoret. Comput. Sci.*, 15:1–25, 1981.
28. Gunther Schmidt. Programs as partial graphs II: Recursion. *Theoret. Comput. Sci.*, 15:159–179, 1981.
29. 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.
30. 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.

31. Gunther Schmidt. Theory Extraction in Relational Data Analysis. In de Swart et al. [14], pages 68–86. 273 pages.
32. Gunther Schmidt and Thomas Ströhlein. Relationen, Graphen und Strukturen, 1975. Internal Report.
33. Gunther Schmidt and Thomas Ströhlein. *Relationen und Graphen*. Mathematik für Informatiker. Springer-Verlag, 1989. ISBN 3-540-50304-8, ISBN 0-387-50304-8.
34. Gunther Schmidt and Thomas Ströhlein. *Relations and Graphs — Discrete Mathematics for Computer Scientists*. EATCS Monographs on Theoretical Computer Science. Springer-Verlag, 1993. ISBN 3-540-56254-0, ISBN 0-387-56254-0.
35. Ingrid Taferner. Untersuchungen von Aussagen der dynamischen Logik auf relationenalgebraischer Grundlage, 1982. Diploma thesis at *Technische Universität München* supervised by Gunther Schmidt in co-operation with Rudolf Berghammer.
36. Alfred Tarski and Steven R. Givant. *A Formalization of Set Theory without Variables*, volume 41 of *Colloquium Publications*. American Mathematical Society, 1987.
37. Gottfried Tinhofer and Gunther Schmidt, editors. *Graph-Theoretic Concepts in Computer Science*, volume 246 of *Lect. Notes in Comput. Sci.* Springer-Verlag, 1987. Proc. 12$^{\text{th}}$ Int'l Workshop WG '86, Kloster Bernried, Jun 17–19, 1986, ISBN 3-540-17218-1, ISBN 0-387-17218-1.
38. Michael Winter. Generating Processes from Specifications Using the Relation Manipulation System RelView. *Electronic Notes in Theoretical Computer Science*, 44(3):27 pages, 2003.
39. Hans Zierer. Relationale Semantik, 1983. Diploma thesis at *Technische Universität München* supervised by Gunther Schmidt in co-operation with Rudolf Berghammer.
40. Hans Zierer. Programmierung mit Funktionsobjekten: Konstruktive Erzeugung semantischer Bereiche und Anwendung auf die partielle Auswertung. Dissertation, 1988.
41. Hans Zierer. Relational algebraic domain constructions. *Theoret. Comput. Sci.*, 87:163–188, 1991.

Journal on Relational Methods in Computer Science, Vol. 1, 2004, pp. 314 - 338
Received by the editors March 21, 2004, and, in revised form, September 21, 2004.
Published on December 10, 2004.

This article may be accessed via WWW at `http://www.jormics.org`.