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
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 assistant 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 \texttt{Ascii}\-form. Opposed to these external translations, also internal ones shall be supported, namely those translating a relational formula in component-free 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/non-continuous 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 so-called construc-
tors 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 formu-
lating 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.

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. $\text{VectToElem}$, e.g., provides the transition from a vector to the corresponding element in the powerset, while $\text{RelaToVect}$ converts a relation to the corresponding vector in the direct product.
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 :\lor: and :*: are defined using the other operations. The first resembles the often discussed fork operator \(\nabla\), 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 \( \text{T}_{\text{E}}\text{X} \) as
\[
\langle X \rightarrow R \cup X.X \rangle \quad \langle X_{O_1,O_2} \rightarrow R_{O_1,O_1} \cup X_{O_1,O_1}:X_{O_1,O_1} \rangle
\]
\[
\langle X \rightarrow R \cup X.X^T.X \rangle \quad \langle X_{O_1,O_2} \rightarrow R_{O_1,O_2} \cup X_{O_1,O_2}:X_{O_1,O_2}^T:X_{O_1,O_2} \rangle
\]

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.

\[
data ElemSET \ = \ VarES \ String \ CatObject \ |
  ES \ ElemVari \ [Formula] \ | \ EX \ [ElemTerm] \ CatObject
\]
\[
data VectSET \ = \ VarVS \ String \ CatObject \ |
  VS \ VectVari \ [Formula] \ | \ VX \ [VectTerm] \ CatObject
\]


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 \( RX [] 01 02 \).

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

```haskell
leastFixedPoint fctl1 =
  let ([],[],[],[rv],[]) = supply 999 0 0 0 1 0
  rt = RV rv
  relaSet = RS rv [RF $ RFctAppl fctl1 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.

```haskell
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 \( f = \text{Disjunct} g \ h \), it will be handled as a formula: \( \text{Negated} f \). Until that point, i.e. as long as negation is something like \( A \subseteq / B \), the type is a convenient way of correctness control. In addition, it allows pattern matching.
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 $\text{VE } vt \ et$ 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 $\text{REE } rt \ et1 \ et2$ 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 $\text{Bool}$; 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.

```haskell
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.

```haskell
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 \( \text{dom}, \text{cod}, \text{typeOf} \). The typical checks are provided for well-formedness using \( \text{isWellFormed} \).

```haskell
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 \( \text{cod } A = \text{dom } B \). 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 \( \text{Nothing} \) is returned. If they are equal, \( \text{Just } [] \) is returned. In other cases, \( \text{Just} \) 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.

```haskell
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 \texttt{Ascii}-text is some sort of a translation.

4.1 Translation into \texttt{TeX}

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

```haskell
class Tex a where
  tex :: Bool -> a -> String
```

with \texttt{tex} allowing to transfer constructs automatically to \texttt{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.,

\[ A \subseteq B \quad \text{as opposed to} \quad \forall x, y : (x, y) \in A \rightarrow (x, y) \in B \]

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 interpret 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 interpret 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.

```haskell
data InterpretObjs = Carrier Carrier CatObject Int
data InterpretCons = InterCon InterCon ElemConst Int
data InterpretVect = InterVec InterVec VectConst [Bool]
data InterpretRela = InterRel InterRel RelaConst [[Bool]]
data InterpretFunc = InterFct InterFct FuncConst [Int]
data InterpretVFct = InterVFc InterVFc VectFct ([Bool] -> [Bool])
data InterpretRFct = InterRFc 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:

```haskell
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.

```haskell
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.

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

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 \(\Pi\), \(\Rho\), \(\Iota\), \(\Kappa\), \(\Epsilon\) have not been mentioned here. Interpretations for these are again generated automatically by the system as shown by the following 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.

![Diagram of direct sum]

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.

\[
\begin{align*}
\iota; \iota^T &= \mathbb{I}, \\
\kappa; \kappa^T &= \mathbb{I}, \\
\iota; \kappa^T &\subseteq \mathbb{I}, \\
\iota^T; \iota &\cup \kappa^T; \kappa = \mathbb{I} \\
\end{align*}
\]

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

\[
\begin{align*}
\forall x : \left( \forall y : \left( \exists u : (x, u) \in \iota \land (y, u) \in \iota \right) \implies x = y \right) \land \\
\forall x : \left( \exists v : (x, v) \in \iota \right), \\
\forall x : \left( \forall y : \left( \exists u : (x, u) \in \kappa \land (y, u) \in \kappa \right) \implies x = y \right) \land \\
\forall x : \left( \exists v : (x, v) \in \kappa \right), \\
\forall x : \left( \forall y : \left( \exists u : (x, u) \in \iota \land (y, u) \in \kappa \right) \right), \\
\forall x : \left( \forall y : \left( \exists u : (u, x) \in \iota \land (u, y) \in \iota \right) \lor \\
\exists u : (u, x) \in \kappa \land (u, y) \in \kappa \implies x = y \right) \land \\
\forall x : \left( \exists v : (v, x) \in \iota \lor \exists v : (v, x) \in \kappa \right)
\end{align*}
\]

In our language an equational universal characterisation may be formulated. Two category objects are bound together using two injective mappings \(\iota, \kappa\) satisfying the following formulae

\[
\text{sumCharacterizingFormulae o1 o2} = \\
\text{let } \iota = \text{Iota o1 o2} \\
\text{     kappa = Kappa o1 o2} \\
\text{     iotaT = Conv o1 o2} \\
\text{     kappaT = Conv o1 o2} \\
\text{     iiT = iota :***: iotaT} \\
\text{     kkT = kappa :***: kappaT} \\
\text{     iTi = iotaT :***: iota)}
\]
A Multilevel Relational Reference Language

\[\begin{align*} 
\kappa T &= \kappa : \text{null to} : \kappa T \\
\iota T &= \iota : \text{null to} : \kappa T \\
in \{ RF \# i1T : \text{null to} : \text{Ident o1}, \\
RF \# k1T : \text{null to} : \text{Ident o2}, \\
RF \# i1T : \text{null to} : \text{NullR o1 o2}, \\
RF \# i1T : \text{null to} : \text{Ident (DirSum o1 o2)} \}
\end{align*} \]

\[\text{sumTheory o1 o2} = \text{TH "Sum-Theory" [o1, o2] [] [] [] [] [] []}} \]

(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 \(\pi, \rho\) are used satisfying in a long or short form

\[\begin{align*} 
\pi_1 \times \pi_2 &:= \pi \quad \pi_1 \times \pi_2 = \pi_1 \\
\rho_1 \times \rho_2 &:= \rho \quad \rho_1 \times \rho_2 = \rho_2
\end{align*} \]

Universal characterisation of the direct product

\[\begin{align*} 
\pi_1 : \pi = \emptyset, & \quad \pi_1 : \pi_1 \times \pi_2, \pi_1 \times \pi_2 = \pi_1, \\
\rho_1 : \rho = \emptyset, & \quad \rho_1 : \rho_1 \times \rho_2, \rho_1 \times \rho_2 = \rho_2,
\end{align*} \]

\[\begin{align*} 
\mathcal{I} \subseteq \pi_1 : \rho, & \quad \mathcal{I} \supseteq \pi_1 \times \rho_2, \rho_1 \times \rho_2 \supseteq \pi_1 \times \rho_2, \\
\pi_1 \times \rho_2 \cap \rho_1 : \rho_1 = \emptyset & \quad \pi_1 \times \rho_2 \cap \rho_1 \times \rho_2 = \emptyset
\end{align*} \]

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

\[\begin{align*} 
\forall x : \exists y : \{ \exists u : (u, x) \in \pi \land (u, y) \in \pi \} & \implies x = y \\
\forall x : \exists y : \{ \exists v : (v, x) \in \pi \}
\end{align*} \]
332  Gunther Schmidt

\[\forall x : \langle \forall y : \exists u : (x, u) \in \pi \land (y, u) \in \pi \rangle \land \langle \exists u : (x, u) \in \rho \land (y, u) \in \rho \rangle \implies x = y \rangle \land \langle \forall x : \langle \exists v : (x, v) \in \pi \rangle \land \langle \exists v : (x, v) \in \rho \rangle \rangle\]

\[
\text{prodCharacterizingFormulae o1 o2 =}
\text{let ppi = Pi o1 o2}
\text{rho = Rho o1 o2}
\text{ppiT = Convs ppi}
\text{rhoT = Convs rho}
\text{ppippiT = ppiT :***: ppi}
\text{rhoTrho = rhoT :***: rho}
\text{in [RF $ ppiTppi :===: Ident o1,}
\text{RF $ rhoTrho :===: Ident o2,}
\text{RF $ UnivR o1 o2 :<==: (ppiT :***: rho),}
\text{RF $ ppippiT :&&&: rhorhoT :===: Ident (DirPro o1 o2)]}
\]

\[
\text{prodTheory o1 o2 = TH "Prod-Theory" [o1,o2] [] [] [] [] [] [prodCharacterizingFormulae o1 o2]}
\]

6.3 Characterization of Direct Powers

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

\[\text{syq}(\epsilon, \epsilon) \subseteq \Pi, \quad \text{syq}(\epsilon_O, \epsilon_O) \subseteq \mathbb{I}_{\mathcal{P}(O)}\]

\[\forall v : (\mathbb{I} \subseteq \Pi : \text{syq}(\epsilon, v)), \quad \langle \forall v_O \subseteq \mathcal{P}(O) : \mathbb{I}_{\mathbb{I}_{\mathcal{P}(O)}} \subseteq \mathbb{I}_{\mathbb{I}_{\mathcal{P}(O)}} : \text{syq}(\epsilon_O, v_O) \rangle\]

\[
\begin{tikzpicture}
  \node (O) at (0,0) {$O$};
  \node (X) at (3,0) {$X$};
  \node (P) at (1.5,1.5) {$\mathcal{P}(O)$};
  \node (R) at (1.5,0) {$R$};
  \draw[->] (O) -- (P) node[above] {$\epsilon$};
  \draw[->] (P) -- (X) node[above] {$\text{syQ}(\epsilon, R)$};
  \draw[->] (O) -- (R); \draw[->] (X) -- (R);
\end{tikzpicture}
\]

Universal characterisation of the direct power

\[
\text{powerCharacterizingFormulae o1 =}
\text{let epsi = Epsi o1}
\text{vvv = VarV "v" o1}
\text{al = UnivR UnitOb (DirPow o1)}
\text{syQEpsiEpsi = SyQ epsi epsi}
\text{syQv v = SyQ epsi v v}
\text{in [RF (syQEpsiEpsi :<==: (Ident (DirPow o1))]},
\]
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 even new category object variables are taken and the result is not well-formed.

\[
A_{02,05} \cap B_{03,06} \subseteq (A_{02,05} \cap C_{04,07}) \cdot (B_{03,06} \cap A_{02,05}^\top) \cdot C_{04,07}
\]

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

\[
A \cdot B \cap C \subseteq (A \cap C \cdot B^\top) \cdot (B \cap A^\top \cdot C)
\]
\[
A_{01,02} \cdot B_{02,03} \cap C_{01,03}
\subseteq (A_{01,02} \cap C_{01,03}) \cdot (B_{02,03} \cap A_{01,02}^\top) \cdot C_{01,03}
\]

As a corresponding example we now show the Schröder rules.
Gunther Schmidt

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 TpX-representations with and without typing are

\[ A \subseteq B \iff \overline{A} \subseteq \overline{B} \]
\[ A_{01,02} \subseteq B_{02,03} \iff A_{01,02}^{\overline{1}_1} \subseteq B_{02,03} \]
\[ A \subseteq C \iff \overline{C} \subseteq \overline{A} \]
\[ A_{01,02} \subseteq B_{02,03} \iff \overline{A_{01,02}} \subseteq \overline{B_{02,03}} \]

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

\[
\begin{align*}
\frac{\cup xP \cup Qy}{xP \cup Qy} \\
\frac{xP \circ y \circ xQy}{xP \circ y \circ xQy}
\end{align*}
\]

\[
\frac{xP \circ y \circ xP \circ y}{xP \circ y \circ xP \circ y} \quad | \quad pQy \circ xP \circ y
g\]

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.

\[
DC = \overline{C}
\]

\[
\begin{align*}
DC \subseteq \overline{C} \quad | \quad \overline{C} \subseteq DC
\end{align*}
\]

\[
\begin{align*}
xDC \circ y \circ x \circ y
\end{align*}
\]
One will easily recognize that the vertical bar between the subtrees means \textit{and}. The variables $x, y$ which appear when switching from the relational to the element-wise consideration are universally quantified where comma-separation means \textit{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 comprehensible form.

8 \hspace{1em} \textbf{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: \textit{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 \texttt{Isabelle} system to have even superior possibilities in theorem proving. Thirdly, there will be some student paper to reengineer the former \texttt{RALF} system according to these new standards. As a fourth point, we aim at triggering the \texttt{RELView} machine out of this language using its \texttt{Kure} interface. As a fifth point connection to the \texttt{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

12. Rudolf Berghammer and Hans Zierer. Relational algebraic semantics of deterministic and nonde-

13. Rodrigo Cardoso. Untersuchungen von parallelen Programmen mit relationenalgebraischen Meth-


Received by the editors March 21, 2004, and, in revised form, September 21, 2004.
Published on December 10, 2004.
Permission to copy for private and scientific use granted.
This article may be accessed via WWW at http://www.jormics.org.