Departamento de Informática

Universidade do Minho

Department of Computer Science
University of Minho

Campus de Gualtar,
4700 Braga
Portugal

Tel: +351-53-604470
Fax: +351-53-604471

Formal Methods Group

TECH. REPORT #UMDITR9702

C. J. Rodrigues and J. N. Oliveira

Normalization “is” Data Reification

Normalization “is” Data Reification

C.J. Rodrigues and J.N. Oliveira ⋆

Dep. Informática,
Universidade do Minho,
Campus de Gualtar, 4700 Braga,
Portugal

Abstract. This report presents a constructive, calculational approach to relational database normalization theory. The main corpus of the report is devoted to showing that relational database model theory can be regarded as a subset of SETS, a data reification calculus based on set-theory and (a modest use of) category theory.

We present a new set of SETS laws which prevent from the violation of normal forms caused by partial dependencies, transitive dependencies and multivalued dependencies. These are isomorphism laws which eliminate a “semantic” datatype invariant and induce a “concrete” data invariant in the reification process corresponding to the normalization task taking place.

The scope of SETS is much wider than the relation model and extends to recursive datatypes, imperative programming *etc.*However, database design has been so far one of its most relevant areas of practical application, including industrial information system design.

1 Context

In the early information system era data were stored in unconnected files. This approach was likely to introduce redundancy and raise consistency problems. The *data-base* concept [28] emerged as a natural evolution providing for an integrated approach to information engineering.

Data bases have an underlying *data model* which defines the way in which information is to be structured. One of the first data models adopted in the past by database management systems was the so-called *hierarchical model*, in which information was organized in hierarchies (“forests” or collections of trees) the nodes of which represented entity (record) sets. The basic operation on a hierarchical database is the so-called *tree walk*, which is unidirectional (from ascendants to descendants). This strategy was found to be too restrictive and was overcome by introducing direct connections that go up a given tree or even from one tree to another tree in the forest. The underlying hierarchy is thus transformed into a graph, or a *network*, a step which explains the subsequent evolution from the hierachical model to the so-called *network model* of information [28].

In the network model, information is modelled by a directed graph (network), where nodes represent entity sets and arcs many-to-one relationships. A node is called a *logical record type*. This is a set of logical records which share the same logical record format. The names of the individual components of a logical record format are called *fields*. An arc is called a *link*. A link between two logical record types A and B, from A to B, represents a many-to-one relationship between A and B. The main operations on a network database are *selection* on a given logical record type and the *navigation* on the network, *i.e.* the follow up of links in any direction along the network. The CODASYL database system is perhaps the most significant example of a network database. Its original proposal was made in 1971 [1].

The first theoretical contributions to a new data model paradigm, the so-called *relational data model*, were published in the early seventies, among which the contributions of Codd [2–5] were outstanding.

The information system community is indebted to Codd for his pioneering work on the very foundations of the *relational data model* theory [2]. Relational algebra was defined in [6], which includes operators from set theory and specific operators from the relational model. (Some of the relational operators were already introduced in [2].) The concept of a *functional dependency* is already present in [2], in the form of keys. Functional dependencies that do

⋆ Dep. Informática, Universidade do Minho, Campus de Gualtar, 4700 Braga, Portugal. EMAIL: {cjr, jno}@di.uminho.pt.

not follow from keys were introduced in [5], for the purpose of *normalization*, that is, of information redundancy elimination. In [4, 5] the so-called second and third *normal forms* of relational databases were introduced. Finally, the so-called *Boyce-Codd normal form* was introduced in [7].

The success of the *relational model* was immediate and overwhelming, both in academic and commercial database development. In 1983 a book by Maier [12] was published which became a standard in the literature on the mathematics of relational database design.

2 Motivation

The relational data model surprises for its simplicity and easy-to-understand formal basis, which explains its preponderance in the current days. This results from its uniformity when compared with the antecessor models, as Maier [12] explains:

> *One of the major advantages of the relational model is its uniformity. All data is viewed as being stored in tables, with each row in the table having the same format. Each row in the table summarizes some object or relationship in the real world.*

However, with the advent of *formal methods* [9, 10, 27], software desgin theorists have become very sensitive to accuracy, rigour and formal correctness in the software production process. The fact that there is far too much intuition and craft in overall software production is well-known, which becomes counter-productive in the sense that code is hardly reused, it is difficult to understand, it is much dependent on authorship, *etc.*

The main target of the formal methods trend is to drive software production into good engineering standards. As a rule, these methods split software production into a *specification phase*, in which a mathematical model is built of the contractor requirements, followed by an *implementation phase* in which such model is somehow converted into a runnable software artifact.

This decade has withnessed advances in the research on the mathematics of program design and on formal methods which show that implementations can be effectively *calculated* from specifications [16, 20, 13]. So, in a sense, software technology is becoming a mature discipline in its (however late) adoption of the "universal problem solving" strategy which one is taught at school:

- understand your problem
- build a mathematical model of it
- reason in such a model
- upgrade your model, if necessary
- calculate a final solution and implement it,

the sophistication of which is only dependent upon the underlying mathematics — from simple arithmetics at primary school to systems of liner equations, then to differential/integral equations and now, in software calculi, to systems of (recursive) equations on domain spaces.

The relational model no doubt provided a formal, yet pragmatic standard for information system problem solving. However, a few questions have since been on demand, which we attempt to address in this report: if relational database theory is a formal method for relational database design, is it sufficiently "formal"? If so, is it a specification theory or a reification one? Can it be regarded as a *program calculus*?

The main corpus of this report is devoted to showing that relational database model theory can be regarded as a subset of the SETS [20, 21] reification calculus. SETS arose in 1987 [19] as an attempt to introduce explicit transformational design rules in program data structuring. In the curricula of the software design courses at the authors' university, the SETS calculus has been offered since 1988 as an alternative to conventional relational database design. The original framework of [19] consolidated in [20] and [21]. It was later extended with a SETS semantics for ERA (Entity-Relationship-Attribue) diagrams [24] and, at the same time, with the main results of this report, which became available in 1993 in the master's thesis of one of the authors [25]. By far, the most comprehensive account of the calculus is an as yet unpublished text book, currently in its 4th edition [22].

3 Structure of this document

Section 4 presents a brief introduction to the relational database model, its aim solely being to identify the main concepts and terminology that will be used throughout the report. Well-known books in the literature (such as *e.g.* [12]) will provide the reader with the many notions omitted here for economy of presentation.

Section 6 presents the SETS calculus in some detail. The main body of this report consists of section 8, in which normal form theory is re-interpreted as a SETS subcalculus. In the last section, some conclusions and directions for future research are presented.

4 Normalization — A Brief Overview

Since its early days [14], relational database design starts from the so-called *database relation scheme* definition. Since relational databases are collections of n-ary relations, such a scheme, consisting in a set of *attribute names*, records the common format of all the elements (tuples) of each individual relation. For instance, an airline schedule might be described by a relation with scheme:

$$\begin{aligned} R = \{ & \\ & \#FLIGHT, ORIGIN, \\ & DESTINATION, DEPARTURE, \\ & ARRIVAL, DURATION \\ \} & \end{aligned} \tag{1}$$

Concerning each relation scheme, there may be explicitly listed subsets of its attribute-name set (the so-called "designated keys") which uniquely identify the tuples of relations with that scheme, that is, which exhibit the "property of key". For example, in relation scheme R above (Eq.(1)) it seems reasonable to single out the following designated keys:

$$\begin{array}{l} \{\#FLIGHT\} \\ \{ORIGIN, DESTINATION, DEPARTURE\} \\ \{ORIGIN, DESTINATION, ARRIVAL\} \end{array} \tag{2}$$

Designated keys with a minimum number of attributes are simply called *keys*. By contrast, sets of attributes containing more than one key are called *superkeys* [1]. For example, all the designated keys listed in above in Eq.(2) are keys.

Designated keys are a way of prescribing *semantics* to relational data in the sense that they impose restrictions to the admissible relations with a certain scheme. So-called *functional dependencies* are generalizations of (designated) keys. Returning to the example, one might find it important to impose the following restriction to any relation with scheme R: *To a given flight, with a certain departure and arrival, corresponds a well determined duration.* That is to say, *a given flight, identified by its departure and arrival times, has one and only one duration.*

This restriction is an example of a functional dependency which one would state more formally as folows: *attribute $DURATION$ is functionally dependant on $DEPARTURE$, $ARRIVAL$.* This is normally abbreviated by

$$DEPARTURE\ ARRIVAL \rightarrow DURATION$$

which can also be read as follows: *$DEPARTURE\ ARRIVAL$ functionally determines $DURATION$.*

The formal definition of a *functional dependency* follows.

Definition 1. *Given subsets $X, Y \subseteq R$ of the relation scheme R of a relation r, this relation is said to satisfy a given functional dependency $X \rightarrow Y$ iff all pairs of tuples $t, t' \in r$ which "agree" on X also agree on Y, that is,*

$$\forall t, t' \in r : (t\,X = t'\,X) \Rightarrow (t\,Y = t'\,Y)$$

□

[1] Superkeys verify the property of key but their number of attributes can be reduced without affecting that property.

In relational database design practice, starting relation schemes tend often to be *too large*, *unstructured* and affected by functional dependencies. The goal of *normalization* is to find a set of relation schemes — also called a *database scheme* — which is free of information redundancy (*i.e.* "is normalized") and free of explicit functional dependencies, and yet able to represent the same information as the initial relation scheme.

The first step of standard normalization consists of eliminating the structure of the relation scheme, *i.e.* of decomposing attributes with structured domain. Each time this first step is applied, the relevant relation scheme is decomposed into two or more relation schemes. The result is a relation scheme which is said to be in the *first normal form* (1NF).

Normalization proceeds via one of two alternative procedures: normalization through *decomposition* or normalization through *synthesis*.

The first option is chosen more often. Proceeding by relation scheme decomposition, the aim is to eliminate "undesirable" properties, namely those which contradict the aphorism: *every attribute which does not occur in the relation scheme key should depend on the key, the whole key and nothing but the key.*

The second step consists of eliminating partial dependencies on the key in order to attain the so-called *second normal form* (2NF). The concept of a *partial dependency* is defined as follows:

Definition 2. *Let $X \subseteq R$ be a subset of relation scheme R and A an attribute in R. Should functional dependency $X \to A$ be reducible to $X' \to A$, for $X' \subseteq X$, then A will be said to be* partially dependant *on X.* $\Box$

Finally, the third step eliminates transitive dependencies on the key. The concept of *transitive dependency* is defined thus:

Definition 3. *Let A be an attribute which occurs in functional dependencies $X \to X'$ and $X' \to A$. Admit also that the inverse dependency $X' \to X$ is not satisfied and A is not a member of either X or X'. Under these conditions, A is said to be* transitively dependant *on X.* $\Box$

Transitive dependencies often correspond to functional dependencies involving attributes not occuring in the key. By eliminating them one obtains a database scheme which is said to be in the so-called *third normal form* (3NF), usually referred to simply as *normalized form*.

The notion of decomposition was explained so far admitting that the relation scheme has a single key. In the general case, one has to admit the possibility of many different keys. The two latter normalization steps above are easily generalized to this case, by introducing the concept of a *nonprime* attribute, that is, an attribute that does not occur in any key.

Normalization via decomposition has some well-known inconveniences, namely: it does not enforce the absence of explicit functional dependencies, and it does not enforce a database scheme with a minimum number of relation schemes.

The so-called *synthesis algorithm* [12] is free of these problems. However, known objections to its practical usefulness are: it is less intuitive, and it is not gradual.

There are further normal forms which, in practice, are not so often considered in database design. Such is the case of the so-called *Boyce-Codd normal form* and *fourth normal form* (4NF). The latter involves another type of dependencies: *multivalued dependencies*. This concept of a *multivalued dependency* is defined informally as follows (see *e.g.* [28]):

> Suppose we are given a relation scheme R, and X and Y are subsets of R. Intuitively, we write $X \twoheadrightarrow Y$ (read "X *multidetermines* Y" or "there is a multivalued dependency of Y on X") if, given values for the attributes of X, there is a set of zero or more associated values for the atributes of Y, and this set of Y-values is not connected in any way to values of the attributes in $R - X - Y$.

This leads to the formal definition which follows.

Definition 4. *Multivalued dependency $X \twoheadrightarrow Y$ is satisfied wherever, for each pair of tuples which agree within X,* i.e. *such that*

$$t_1\, X = t_2\, X$$

there is, in the same relation, another pair of tuples t_3 and t_4 such that:

(a) $t_3\,X = t_4\,X = t_1\,X = t_2\,X$
(b) $t_3\,Y = t_1\,Y$
$t_3(R - XY) = t_2(R - XY)$
(c) $t_4\,Y = t_2\,Y$
$t_4(R - XY) = t_1(R - XY)$

$\square$

5 Program Calculation versus Relational Design

Despite its enormous popularity as a database development method, the relational model suffers from some drawbacks with respect to certain parameters generally reputed as desirable in software development methods:

- Lack of abstraction — if we consider the initial relational scheme, the attribute domains and the functional dependencies associated to the relation scheme as the syntactical and semantical components of a problem specification, this specification is beforehand compromised with a particular implementation: a relational ("tabular") implementation.
- Little expressive power — the specification is poor in terms of the definition of semantic properties for the relations with a certain relation scheme. Reference [24] provides an illustration of semantically important properties of a design which cannot be captured by functional dependencies and other types of dependencies.
- Data decomposition — one may ask: where is the *reification* strategy of the method? Is normalization a reification process? Is it rigorous enough?
- Algorithmic decomposition — there is no stepwise operation reification. Operations are defined directly at the relational implementation level.

In brief, the relational model alone is insufficient as a generic software design methodology. The main objective of this report is the study of data reification towards a relational database environment. In this context, standard relational calculus will be re-interpreted in terms of data reification, as understood in the SETS calculus. In particular, the following points will be addressed:

- Is the former a subcalculus of the latter?
- In what extent is normalization a reification strategy?
- In the SETS calculus one easily calculates simple functional dependencies. What about other types of dependencies?

6 A Synopsis of the SETS Notation

The SETS notation and calculus [20, 21, 23] can be regarded as a categorial re-interpretation of conventional model-oriented specification as understood in the VDM [10] or Z [27] tradition.

SETS stands for "**Set**-based specification", where $\mathbf{Set}$ is the category of functions on (countable) sets. It is a calculus of *abstraction invariants* as understood in the following quotation from [16]:

> *The laws of Section 17.4 dealt with a very general case of data refinement, in which the coupling invariant linking the abstract and concrete states could be anything whatever.*
> *This is actually a fairly rare occurrence in everyday program development however: it is much more common for the coupling invariant to be functional from concrete to abstract.* [...]
> *The general form for such coupling invariants, called* functional abstractions *is*
> $$a = \mathsf{af}\ c \wedge \mathsf{dti}\ c\ , \qquad (17.1)$$
> *where* af *is a function, called the* abstraction function *and* dti *is a predicate, in which a does not appear, called the* data-type invariant. [...]

But instead of writing and reason about predicates such as *(17.1)*, in predicate logic, inequations of the form

$$A \trianglelefteq_{\mathsf{af}} B_{\mathsf{dti}} \tag{3}$$

are recorded meaning that map $A \xleftarrow{\mathsf{af}} B_{\mathsf{dti}}$ exists and is surjective, where $B_{\mathsf{dti}} \stackrel{\rm def}{=} \{b \in B \mid \mathsf{dti}\, b\}$.

As a specification language, SETS should not be regarded as a competitor to VDM or Z, but rather as a framework to write and reason about specifications in a simple, "pure" mathematical notation, the one arising from the constructs of the (cartesian closed) category of sets — product ($A \times B$), exponentiation (A^B) and co-product ($A + B$). Product and exponentials are defined in the usual manner,

$$A \times B \stackrel{\rm def}{=} \{(a, b) \mid a \in A \wedge b \in B\}$$
$$A^B \stackrel{\rm def}{=} \{f \mid f : B \to A\}$$

Co-product (disjoint union) is handled explicitly by using integer tags, *i.e.*

$$A + B = (\{1\} \times A) \cup (\{2\} \times B)$$

On top of this, in SETS one can build data models using the following *derived* constructs:

- subsets of a finite set A (*cf.* $\mathsf{set\ of}\ A$ in VDM):
$$2^A$$
- binary relations over finite sets A and B (*cf.* $\mathsf{set\ of}\ (A \times B)$ in VDM):
$$2^{A\times B} \tag{4}$$
- partial maps from finite A to B (*cf.* $\mathsf{map}\ A\ \mathsf{to}\ B$ in VDM):
$$A \rightharpoonup B = \bigcup_{K \subseteq A} B^K$$
- finite sequences on a set A (*cf.* $\mathsf{seq\ of}\ A$ in VDM):
$$A^\star = \bigcup_{n \geq 0} A^{\overline{n}}$$
where each exponent $\overline{n}$ denotes the initial segment of $I\!\!N$ whose cardinality is n. For simplicity, we will write n instead of $\overline{n}$, and therefore 0 instead of the empty set $\emptyset$ [2].
- union types or "pointers" (*cf.* $[A]$ in VDM):
$$A + 1$$
where 1 stands for any singleton set.

A last ("meta") construct is recursive definition,

$$X \cong \mathsf{F}\, X$$

where X is the name of a data sort and F is a SETS-expression ("functor") involving the above primitive or derived constructs [3].

Recursive definitions are a powerful mechanism for specifying arbitrarily elaborate data structures, see *e.g.* [21]. SETS's kernel functional notation includes all set-theoretical operators typical of constructive specification languages (such as VDM, Z, *etc.*) dealing with lists ($A^\star$), sets (2^A), mappings ($A \rightharpoonup B$) and so on.

[2] In SETS, all sets with the same cardinality are isomorphic and therefore indistinguishable from a specification point of view. Initial segments of $I\!\!N$ provide an abstract mechanism for specifying enumerated types, of which only cardinality is retained. Particularly common enumerated data types are 0 (the empty set), 1 (any singleton set) and 2 (the Booleans). Should the specification style become too terse, the specifier may adopt "cannonical" standards among each cardinality class, *e.g.* $2 = \{TRUE, FALSE\}$ and $1 = \{NIL\}$ (NIL is a particular good choice concerning union types).

[3] Not every F can participate in the definition of a recursive data domain. See [20] for a discussion on this topic.

7 How SETS Approaches Data Refinement

As we have seen above in (3), a SETS inequation of the form

$$A \trianglelefteq_f B_\phi \tag{5}$$

records the fact that map $A \xleftarrow{f} B_\phi$ exists and is surjective, where $B_\phi \stackrel{\mathrm{def}}{=} \{b \in B \mid \phi\, b\}$. Moreover, we may read (5) in the following way:

- there is an *abstraction surjection* f from B to A;
- f is a partial function, and ϕ is the characteristic predicate defining its domain — the so-called *concrete invariant* induced in B.

We may also say that A is implemented by B or that B is more "redundant" than A. In the context of the above quotation from [16], equation (5) states that a functional abstraction invariant holds between the abstract variables in A and the concrete variables in B.

7.1 The Inequational Basis of SETS

The SETS refinement calculus is a collection of inequations with the form of (5). It can thus be regarded as an (inequational) calculus of (functional) *abstraction invariants*, as understood by [16]. As a rule, superscript ϕ is omitted wherever $\phi\, b = TRUE$. As a sample, we present three basic SETS-inequational laws:

$$A \rightharpoonup (B \times C) \trianglelefteq (A \rightharpoonup B) \times (A \rightharpoonup C) \tag{6}$$

$$A \rightharpoonup D \times (B \rightharpoonup C) \trianglelefteq (A \rightharpoonup D) \times ((A \times B) \rightharpoonup C) \tag{7}$$

$$A \rightharpoonup B \trianglelefteq 2^{A \times B} \tag{8}$$

which entail the following standard abstraction invariants (see appendix A for the definition of the implied primitive functions and predicates):

$$(6)\begin{cases} f \stackrel{\mathrm{def}}{=} \bowtie \\ \phi \stackrel{\mathrm{def}}{=} eqd \end{cases} \quad (7)\begin{cases} f \stackrel{\mathrm{def}}{=} \bowtie_n \\ \phi \stackrel{\mathrm{def}}{=} dpi \end{cases} \quad (8)\begin{cases} f \stackrel{\mathrm{def}}{=} mkf \\ \phi \stackrel{\mathrm{def}}{=} fdp \end{cases}$$

It can be shown that $\trianglelefteq$ is a pre-ordering on SETS data structures and that set-theoretical isomorphism is the symmetric closure of $\trianglelefteq$, that is, if $A \trianglelefteq_f B$ and $B \trianglelefteq_{f^{-1}} A$ hold, then $A \cong B$ holds, for f a bijection. So, all $\mathbf{Set}$-isomorphisms are laws of the SETS calculus. Among these, which are well-known, we select the following, which will be relevant in the sequel:

$$A \times B \cong B \times A \tag{9}$$

$$A \rightharpoonup 1 \cong 2^A \tag{10}$$

The relevant abstraction functions are bijective and so their inverses exist ($\phi = \lambda b.TRUE$ is omitted):

$$(9)\begin{cases} f \stackrel{\mathrm{def}}{=} \langle \pi_2, \pi_1 \rangle \\ f^{-1} \stackrel{\mathrm{def}}{=} f \end{cases} \quad (10)\begin{cases} f \stackrel{\mathrm{def}}{=} set2fm \\ f^{-1} \stackrel{\mathrm{def}}{=} dom \end{cases}$$

Transitivity of $\trianglelefteq$ means that one can chain $\trianglelefteq$-steps and synthesize overall abstraction maps and invariants. For a chain of n $\trianglelefteq$-steps,

$$\trianglelefteq_i \begin{cases} f_i \\ \phi_i \end{cases}$$

the overall abstraction map and invariant are given by:

$$\trianglelefteq \begin{cases} f \stackrel{\mathrm{def}}{=} \bigcirc_{i=1}^{n} f_i \\ \phi \stackrel{\mathrm{def}}{=} \lambda x.\ \bigwedge_{i=n}^{1} \phi_i((\bigcirc_{j=i+1}^{n} f_j)(x)) \end{cases} \tag{11}$$

For further compositional results which structure SETS calculations on a functorial basis see [21, 22].

8 Expressing Normalization in SETS

Normalization stands at the core of the relational calculus. As we have seen, it basically consists of imposing constraints to data relations which lead to diverse configurations (*normal forms*) with the purpose of avoiding information redundancy.

The SETS calculus can be applied to study normalization from a calculational approach. As we will see shortly, one just needs to encode normal forms in terms of datatype invariants and express data "transformation" across normal forms via the appropriate reification laws.

Relations as Exponentials

As we have seen, the notion of a (binary) relation is expressed in SETS by a powerset (4). This obviously generalizes to n-ary relations,

$$2^{A_1 \times \ldots \times A_n}$$

All the conventional relational operators are expressible via powerset-theoretic operators. For a given tuple $t = (a_1, \ldots, a_n)$ in an n-ary relation ρ, $\pi_i\, t$ denotes the i-th component of "vector" t, that is, $\pi_i\, t = a_i$.

Should complex expressions involving too much nesting of projection π_1, π_2, π_3 *etc.* be found hard to read in practice, one may *rename* them by attaching *attribute names* to each factor in the exponent of a relational declaration:

$$2^{N_1 : A_1 \times \ldots \times N_n : A_n}$$

In this case, $N_1\, t$ could be used instead of $\pi_1\, t$, *etc.*.

Functional dependences as finite mappings

For A, B two given datatypes, one may read "$A \rightharpoonup B$" as the datatype of relations $\sigma \subseteq B \times A$ such that $\sigma \circ \sigma^\circ$ is a subset of identity id_B. In other words, the As are *keys* which uniquely identify the Bs. So, $A \rightharpoonup B$ expresses a functional dependence $A \to B$, recall law (8). Functional dependences can be made explicit by recording them in terms of the fdp relational constraint (86).

8.1 First Normal Form (1NF)

Our first result concerns the first normal form (1NF). Let us first see a toy example. Consider the following specification of a data structure recording information concerning the participants of a conference, congress, symposium *etc.*:

$$\begin{array}{rlll} Participants = \#Part \rightharpoonup & Name & \times & \text{/*name of participant */} \\ & Address & \times & \text{/*address of participant */} \\ & Fee & \times & \text{/*fee */} \\ & MoP & \times & \text{/*chosen method for payment of fee */} \\ & 2^{\#Session} & \times & \text{/*registered for which sessions, } \#Session \text{ = session code */} \\ & 2^{\#Pa} & \times & \text{/*registered for which parallel activities, } \#Pa \text{ = parallel activity code */} \end{array} \quad (12)$$

where $\#Part$, $Name$, $Address$, Fee, MoP, $\#Session$ and $\#Pa$ are atomic types. So, a participant may have applied to various sessions and to various parallel activities — *cf.* $2^{\#Pa}$ and $2^{\#Session}$ in (12), respectively.

It is immediate to see that $Participants$ is not in 1NF, since $2^{\#Session}$ and $2^{\#Pa}$ are non-atomic attributes of a participant. Its conversion to 1NF is obtained via laws which "flatten" these fields of the relational $Participants$ datatype. In detail, we have:

$$\begin{array}{ll} & Participants \\ = & \{ \text{ introducing } X = Name \times Address \times Fee \times MoP \text{ for improved readability } \} \\ & \#Part \rightharpoonup X \times 2^{\#Session} \times 2^{\#Session} \end{array}$$

$$
\begin{array}{ll}
\cong & \{ \text{ law (10) from right to left (twice) } \} \\
& \#Part \rightharpoonup X \times (\#Session \rightharpoonup 1) \times (\#Session \rightharpoonup 1) \\
\trianglelefteq & \{ \ \times\text{-associativity followed by (7) } \} \\
& (\#Part \rightharpoonup X \times (\#Session \rightharpoonup 1)) \times (\#Part \times \#Session \rightharpoonup 1) \\
\trianglelefteq & \{ \text{ again (7) } \} \\
& ((\#Part \rightharpoonup X) \times (\#Part \times \#Session \rightharpoonup 1)) \times (\#Part \times \#Session \rightharpoonup 1) \\
\cong & \{ \text{ (10) twice, now from left to right } \} \\
& ((\#Part \rightharpoonup X) \times 2^{\#Part \times \#Session}) \times 2^{\#Part \times \#Session} \\
\cong & \{ \text{ removing } X \text{ and the parenthesis } \} \\
& (\#Part \rightharpoonup Name \times Address \times Fee \times MoP) \times 2^{\#Part \times \#Session} \times 2^{\#Part \times \#Session} \\
\cong & \{ \text{ introducing } Participants' \ \} \\
& Participants'
\end{array}
$$

Altogether, we have

$$Participants \trianglelefteq Participants' \tag{13}$$

where a single "structured table" in $Participants$ was decomposed into three "flat tables" in $Participants'$ — the first one describes entity $Participant$ while the other two are relationships with other entities. (Hereafter, we sometimes employ the classic relational jargon freely, refering to data domains as attributes, to mappings as tables, *etc.*).

In general, we have the following proposition and lemma:

Proposition 1 (1NF). *The First Normal Form (1NF) is obtained in* SETS *via reification rules (6), (7), (14) and (15), where:*

$$A \rightharpoonup (B + C) \trianglelefteq (A \rightharpoonup B) \times (A \rightharpoonup C) \tag{14}$$
$$A^\star \trianglelefteq I\!N \rightharpoonup A \tag{15}$$

Proof: *See [25].* □

Lemma 1. *The following derived laws can also be used to reach 1NF:*

$$A \rightharpoonup 2^B_+ \cong 2^{A \times B} \tag{16}$$
$$A \rightharpoonup 2^B \trianglelefteq 2^A \times 2^{A \times B} \tag{17}$$
$$A \rightharpoonup C \times 2^B \trianglelefteq (A \rightharpoonup C) \times 2^{A \times B} \tag{18}$$
$$A \rightharpoonup B^\star \trianglelefteq 2^A \times (A \times I\!N \rightharpoonup B) \tag{19}$$
$$A \rightharpoonup C \times B^\star \trianglelefteq (A \rightharpoonup C) \times (A \times I\!N \rightharpoonup B) \tag{20}$$
$$A \rightharpoonup (B \rightharpoonup C) \trianglelefteq 2^A \times (A \times B \rightharpoonup C) \tag{21}$$
$$A \rightharpoonup (\mu X.\mathsf{F}\, X) \trianglelefteq (A \rightharpoonup K) \times (A \times K \rightharpoonup \mathsf{F}\, K) \tag{22}$$
$$A \rightharpoonup B \times (\mu X.\mathsf{F}\, X) \trianglelefteq (A \rightharpoonup B \times K) \times (A \times K \rightharpoonup \mathsf{F}\, K) \tag{23}$$

Proof: *These rules derive from the primitive ones, see [25].* □

Altogether, the rules given so far play an important rôle in derived construct reification, in a relational model context. They can be used to "decompose" complex/nested mappings or sequences into tuples of simpler maps. The structure of the calculated “flat tables” is subject to a datatype invariant induced by the reification process, as exposed in [21]. The semantics implicit in such a datatype invariant complements the identification of the table keys.

It is interesting to note that besides being in 1NF, $Participants'$ (13) is also in 3NF. The attainment of 3NF is not an exclusive of law (18). In fact, it can be checked by inspection that they all garantee 3NF. Not always immediately, as in this example, nor applying the same rule. If, for instance, we had replaced $2^{\#Session}$ by $Session^{\star}$ in $Participants$, we would have to apply law (20) in addition to law (18).

8.2 Second Normal Form

Second normal form (2NF) is characterized in SETS by a rule describing the involved reification strategy:

Lemma 2 (2NF characterization). *Fact*

$$(A \times B \rightharpoonup C \times D)_{\phi} \trianglelefteq_{\bowtie} (A \times B \rightharpoonup D) \times (A \rightharpoonup C) \tag{24}$$

holds, where:

$$\phi(f) \stackrel{\text{def}}{=} \forall x, x' \in dom\, f : \pi_1(x) = \pi_1(x') \Rightarrow \pi_1(f\,x) = \pi_1(f(x')) \tag{25}$$
$$g \bowtie h \stackrel{\text{def}}{=} [x \mapsto \langle h(y), g(x)\rangle | x \in dom\, g \wedge y \in dom\, h \wedge \pi_1(x) = y]$$

Proof: *See [25].* □

A table bearing the structure of the antecedent (*i.e.*, lefthand side) of (24) has $(A\ B)$ as key and satisfies the A *determines* C functional dependency, *cf.* datatype invariant ϕ. It follows that C is partially dependent on $(A\ B)$. C is implicitly *nonprime*, *i.e.* it does not occur in any key. The table is losslessly decomposed (via $\bowtie$) into two tables without nonprime attributes partially dependant on a key.

Corollary 1.

$$A \rightharpoonup B \times C \trianglelefteq_{\bowtie} (A \rightharpoonup B) \times (A \rightharpoonup C)$$

where:

$$g \bowtie h \stackrel{\text{def}}{=} [x \mapsto (g(x), h(y)) | x \in dom\, g \wedge y \in dom\, h \wedge x = y]$$

□

This corollary introduces a reification rule that eliminates the eqd data invariant (84), implicit in rule (6).

Rule (24) introduces redundancy in the process of implementing one “abstract” table by two “concrete” tables. This is not the case when reducing to 2NF. In fact, the relational calculus “implements” the abstract table by a determined pair of tables. Thus, there is a ono to one relationship betwen the abstract table and the pair of tables that implement it. This is better captured by the following lemma.

Lemma 3 (2NF strict characterization). *2NF is strictly characterized by law*

$$(A \times B \rightharpoonup C \times D)_{\phi} \cong_{\psi} ((A \times B \rightharpoonup D) \times (A \rightharpoonup C))/K_{\bowtie} \tag{26}$$

where $K_{\bowtie}$ is the kernel relation induced by $\bowtie$ and

$$\psi : (g, h)/K_{\bowtie} \mapsto g \bowtie h$$

Proof: *Law (26) results from law (24) after the decomposition of the abstraction function $\bowtie$, according to the* Theorem of Function Decomposition.
□

Lemma 4 (2NF concrete domain). *2NF concrete domain may be defined by either of these two isomorphic expressions.*

$$((A \times B \rightharpoonup D) \times (A \rightharpoonup C))/K_{\bowtie} \cong \mu\; ((A \times B \rightharpoonup D) \times (A \rightharpoonup C))_{\gamma} \tag{27}$$

where:

$$\begin{aligned} \gamma(g,h) &\stackrel{\text{def}}{=} \forall x \in dom\, g : \exists y \in dom\, h : \pi_1(x) = y \wedge \\ &\quad\;\; \forall y \in dom\, h : \exists x \in dom\, g : \pi_1(x) = y \\ \mu(g,h) &\stackrel{\text{def}}{=} (g,h)/K_{\bowtie} \end{aligned} \tag{28}$$

Proof: *We will prove that μ is a bijection,* i.e. *a injective and surjective function.*
First, we propose the inverse of μ, μ^{-1}:

$$\begin{aligned} \mu^{-1} : (g,h)/K_{\bowtie} \mapsto\; & (g',h') \\ & \underline{where}\;\; g' = g|\{x \in dom\, g | \exists y \in dom\, h : \pi_1(x) = y\} \\ & \qquad\quad\; h' = h|\{y \in dom\, h | \exists x \in dom\, g : \pi_1(x) = y\} \end{aligned}$$

We begin by proving that μ is injective. By definition, μ is injective iff for any (g,h) in the consequent of (27), fact

$$\mu^{-1} \circ \mu((g,h)) = (g,h)$$

holds.
We have:

$$\begin{aligned} \mu^{-1} \circ \mu((g,h)) &= \mu^{-1}((g,h)/K_{\bowtie}) \\ &= (g',h') \\ &= (g|dom\, g, h|dom\, h) \\ &= (g,h) \end{aligned}$$

as desired.

Next we prove that μ is surjective. By definition, μ is surjective iff for any $(g,h)/K_{\bowtie}$ in the antecedent of (27), fact

$$\mu \circ \mu^{-1}((g,h)/K_{\bowtie}) = (g,h)/K_{\bowtie}$$

holds. We have:

$$\begin{aligned} \mu \circ \mu^{-1}((g,h)/K_{\bowtie}) &= \mu((g',h')) \\ &= (g',h')/K_{\bowtie} \\ &= (g,h)/K_{\bowtie} \end{aligned}$$

as desired.
□

Combining laws (26) and (27) we are lead to a final result.

Proposition 2 (2NF). *Second Normal Form (2NF) is defined in* SETS *by the reification rules corresponding to 1NF and by the specific reification rule:*

$$(A \times B \rightharpoonup C \times D)_{\phi} \cong \bowtie ((A \times B \rightharpoonup D) \times (A \rightharpoonup C))_{\gamma} \tag{29}$$

Proof *Law (29) results from laws (26) and (27), by transitivity (also notice that $\psi \circ \mu = \bowtie$).* □

Example 1 (2NF). In our toy example, the conference database has records with information about the sessions and the concourse of the participants to them. Records also register the rating which each participant assigns to each session he is engaged in.

Suppose that the following starting specification of the records was made:

$$PS = ((\#Session \times \#Part) \rightharpoonup Eval \times (Date \times Hour \times Room \times Theme))_{\varphi} \tag{30}$$

where

$$\varphi(f) \stackrel{\text{def}}{=} \forall x, x' \in dom\, f : \pi_1(x) = \pi_1(x') \Rightarrow \pi_2(f\,x) = \pi_2(f(x'))$$

(undetailed types such as $Hour$, *etc.* must be regarded as atomic.)

The PS datatype is not in 2NF because only factor $Eval$ is fully functionally dependant on the key ($\#Session\ \#Part$). Informally, PS is redundant since, for each participant present in a certain session, the session's information is repeated.

PS is converted to 2NF by application of Proposition 2 after some little preliminary transformation:

$$PS_1 = ((\#Session \times \#Part) \rightharpoonup (Date \times Hour \times Room \times Theme) \times Eval)_{\varphi'}$$

where φ' has the same definition as ϕ, *cf.* (25). The outcome from the application of this Proposition is:

$$\begin{aligned} PS_2 = \ & (((\#Session \times \#Part) \rightharpoonup Eval) \times \\ & (\#Session \rightharpoonup Date \times Hour \times Room \times Theme))_{\gamma} \end{aligned} \tag{31}$$

where γ corresponds to (28).

So, the redundancy was eliminated by the calculation of a new table which records information about sessions only.

Afain, PS_2 is not only in 2NF but also in 3NF.

8.3 Third Normal Form

Third Normal Form (3NF) is characterized in SETS by a rule that describes the reification strategy involved:

Lemma 5 (3NF characterization). *Fact*

$$(B \rightharpoonup A \times C \times D)_{\phi} \trianglelefteq_{\bowtie} (B \rightharpoonup A \times D) \times (A \rightharpoonup C) \tag{32}$$

holds, where:

$$\phi(f) \stackrel{\text{def}}{=} \forall x, x' \in dom\, f : \pi_1(f\,x) = \pi_1(f(x')) \Rightarrow \pi_2(f\,x) = \pi_2(f(x')) \tag{33}$$

$$\begin{aligned} g \bowtie h \stackrel{\text{def}}{=} \ & [x \mapsto \langle \pi_1(g\,x), h(y), \pi_2(g\,x)\rangle | \\ & x \in dom\, g \wedge y \in dom\, h \wedge \pi_1(g\,x) = y] \end{aligned} \tag{34}$$

Proof: *See [25].* □

A table having the structure of the antecedent of (32) has (B) as key and satisfies the functional dependency A *determines* C (*cf.* ϕ). It follows that C is transitively dependant on B. C is implicitly nonprime. The table is losslessly decomposed (*cf.* $\bowtie$) into two tables without nonprime attributes transitively dependant on a key.

As we did with 2NF, we will resort to a more exact result:

Lemma 6 (3NF strict characterization). *3NF is strictly characterized by the law:*

$$(B \rightharpoonup A \times C \times D)_{\phi} \cong_{\psi} ((B \rightharpoonup A \times D) \times (A \rightharpoonup C))/K_{\bowtie}$$

where $K_{\bowtie}$ is the kernel relation induced by $\bowtie$ and

$$\psi : (g, h)/K_{\bowtie} \mapsto g \bowtie h$$

Proof: *Immediate, attend to 2NF.* □

Lemma 7 (3NF concrete domain). *The 3NF concrete domain can be defined by either of these two isomorphic expressions.*

$$((B \rightharpoonup A \times D) \times (A \rightharpoonup C))/K_{\bowtie} \cong \mu\, ((B \rightharpoonup A \times D) \times (A \rightharpoonup C))_{\gamma}$$

where:

$$\begin{aligned}\gamma(g,h) &\stackrel{\mathrm{def}}{=} \forall x \in dom\, g : \exists y \in dom\, h : \pi_1(g\, x) = y \wedge \\ &\quad\ \forall y \in dom\, h : \exists x \in dom\, g : \pi_1(g\, x) = y \\ \mu(g,h) &\stackrel{\mathrm{def}}{=} (g,h)/K_{\bowtie}\end{aligned} \tag{35}$$

Proof: *Immediate, attend to 2NF.* □

Proposition 3 (3NF). *Third Normal Form (3NF) is defined in* SETS *by the reification rules corresponding to 2NF and by reification rule*

$$(B \rightharpoonup A \times C \times D)_{\phi} \cong \bowtie ((B \rightharpoonup A \times D) \times (A \rightharpoonup C))_{\gamma}$$

Proof: *Immediate, attend to 2NF.* □

Example 2 (3NF). Our toy information system takes care of the accommodation for the participants of the conference who asked for help and makes the reservation of the lodgings, for a convenient period, in local hotels. The conference database registers the information relative to hotels and lodgings.

Suppose that the starting specification of the respective record structure was made thus,

$$LH = (\#Part \rightharpoonup di \times Do \times rn \times Hn \times (category \times address))_{\varphi} \tag{36}$$

where di stands for *day in*, Do stands for *day out*, rn stands for *room number*, Hn stands for *hotel name* (these no further detailed types should be regarded as atomic), and where

$$\varphi(f) \stackrel{\mathrm{def}}{=} \forall x, x' \in dom\, f : \pi_4(f\, x) = \pi_4(f(x')) \Rightarrow \pi_5(f\, x) = \pi_5(f(x'))$$

LH is redundant since, for each participant lodged in the same hotel, information relative to the hotel is repeated. This redundancy is expressed by the transitive dependency induced by functional dependency

$$Hn \text{ determines } Category\ Address$$

This transitive dependency violates 3NF. But LH can be converted to 3NF by application of Proposition 3. This one cannot be applied directly to LH, but a preliminary transformation will to:

$$LH_1 = (\#Part \rightharpoonup nh \times (category \times address) \times (di \times Do \times rn))_{\varphi'}$$

where φ' has the same definition as ϕ, *cf.* (33). The outcome is:

$$\begin{aligned}LH_2 = &\ ((\#Part \rightharpoonup Hn \times (di \times Do \times rn)) \times \\ &\ (nh \rightharpoonup category \times address))_{\gamma}\end{aligned} \tag{37}$$

where γ corresponds to (35). The redundancy was eliminated by moving the information relative to hotels to a separate table.

8.4 Boyce Codd Normal Form

Boyce-Codd Normal Form (BCNF) is characterized in SETS by a rule that describes the reification strategy involved:

Lemma 8 (BCNF characterization). *Fact*

$$(D \rightharpoonup A \times B \times C \times E)_\phi \trianglelefteq^{\varphi}_{\bowtie} (D \rightharpoonup A \times B \times E) \times (A \rightharpoonup C) \tag{38}$$

holds, where:

$$\begin{aligned} \phi(f) &\stackrel{\text{def}}{=} \forall x, x' \in dom\, f : \pi_1(f\,x) = \pi_1(f(x')) \Rightarrow \pi_3(f\,x) = \pi_3(f(x')) \wedge \\ &\quad \forall x, x' \in dom\, f : \pi_2(f\,x) = \pi_2(f(x')) \wedge \pi_3(f\,x) = \pi_3(f(x')) \\ &\quad \Rightarrow x = x' \end{aligned} \tag{39}$$

$$\begin{aligned} \varphi(f, g) &\stackrel{\text{def}}{=} \forall x, x' \in dom\, f : \exists y, y' \in dom\, g : \pi_1(f\,x) = y \wedge \pi_1(f(x')) = y' \\ &\quad \Rightarrow \pi_2(f\,x) = \pi_2(f(x')) \wedge g(y) = g(y') \\ &\quad \Rightarrow x = x' \end{aligned} \tag{40}$$

$$\begin{aligned} g \bowtie h &\stackrel{\text{def}}{=} [x \mapsto \langle \pi_1(g\,x), \pi_2(g\,x), h(y), \pi_3(g\,x)\rangle | \\ &\quad x \in dom\, g \wedge y \in dom\, h \wedge \pi_1(g\,x) = y] \end{aligned} \tag{41}$$

Proof: *See [25].*

In this case, a table bearing the structure of the antecedent of (38) has as keys $(B\ C)$ — *cf.* ϕ — and (D), and satisfies functional dependency A *determines* C — *cf.* ϕ again. If follows that C is transitively dependant on D. C is prime since it occurs in a key. The table is losslessly decomposed (*cf.* $\bowtie$) into two tables without any prime attibute transitively dependant on a key. The two tables satisfie a data invariant (*cf.* φ) that requires existence of a key, $(B\ C)$, in the table outcoming from their join.

The SETS reasoning involved in the proof of Lemma 8 deserves to be presented in some detail. First, we will enunciate a general law that will be needed in the sequel.

Theorem 1. *If $A \trianglelefteq^{\phi}_{f} B_\phi$ holds*[4] *and $\alpha : A \longrightarrow 2$ is an extra data invariant over A, then α is "pushed" into B as follows:*

$$A_\alpha \trianglelefteq_f \phi B_{\phi \wedge \alpha \circ f}$$

Proof: *See [11].* □

We recall the rule from 3NF we are interested in:

$$(B \rightharpoonup A \times C \times D)_\beta \trianglelefteq_{\bowtie'} (B \rightharpoonup A \times D) \times (A \rightharpoonup C) \tag{42}$$

where β and $\bowtie'$ have been defined in Lemma 5 as ϕ (*cf.* (33)) and $\bowtie$ (*cf.* (34)), respectively. We instantiate D as $D_1 \times D_2$ in (42). By $\times$-associativity, we get

$$(B \rightharpoonup A \times C \times D_1 \times D_2)_\beta \trianglelefteq_{\bowtie''} (B \rightharpoonup A \times D_1 \times D_2) \times (A \rightharpoonup C) \tag{43}$$

where $\bowtie''$ follows trivially from $\bowtie'$. As $\times$ is commutative (and associative) we next rewrite (43) to:

$$(B \rightharpoonup A \times D_1 \times C \times D_2)_\delta \trianglelefteq_{\bowtie} (B \rightharpoonup A \times D_1 \times D_2) \times (A \rightharpoonup C) \tag{44}$$

[4] $A \trianglelefteq^{\phi}_{f} B_\phi$ is used to mean the same as $A \trianglelefteq^{\phi}_{f} B$.

where δ and $\bowtie$ follow trivially from β and $\bowtie''$, respectively. It happens that $\bowtie$ has the definition given in Lemma 8, *cf.* (41). Finally, we define the data invariant ρ over the antecedent of (44):

$$\begin{aligned}\rho(f) \stackrel{\text{def}}{=}\ & \forall x, x' \in dom\, f : \pi_2(f\, x) = \pi_2(f(x')) \wedge \pi_3(f\, x) = \pi_3(f(x')) \\ & \Rightarrow x = x'\end{aligned} \tag{45}$$

and apply Theorem 1, obtaining law:

$$(B \rightharpoonup A \times D_1 \times C \times D_2)_\phi \trianglelefteq_{\bowtie} ((B \rightharpoonup A \times D_1 \times D_2) \times (A \rightharpoonup C))_\varphi \tag{46}$$

for:

$$\begin{aligned}\phi(f) &= \delta(f) \wedge \rho(f) \\ \varphi(f) &= (\rho \circ \bowtie)(f)\end{aligned}$$

It is easy to check that ϕ and φ have the definition given in Lemma 8, *cf.* (39) and (40), respectively. We can express (46) as:

$$(D \rightharpoonup A \times D_1 \times C \times D_2)_\phi \trianglelefteq_{\bowtie}^{\varphi} (D \rightharpoonup A \times D_1 \times D_2) \times (A \rightharpoonup C) \tag{47}$$

which is reification rule (38) expressed in Lemma 8 up to variable renaming.

Lemma 9 (BCNF strict characterization). *BCNF is strictly characterized by law:*

$$(D \rightharpoonup A \times B \times C \times E)_\phi \cong_\psi (((D \rightharpoonup A \times B \times E) \times (A \rightharpoonup C))_\varphi)/K_{\bowtie} \tag{48}$$

where $K_{\bowtie}$ is the kernel relation induced by $\bowtie$ and

$$\psi : (g, h)/K_{\bowtie} \mapsto g \bowtie h$$

Proof: *Immediate.* □

Lemma 10 (BCNF concrete domain). *BCNF concretre domain may be defined by either of the following two isomorphic expressions:*

$$(((D \rightharpoonup A \times B \times E) \times (A \rightharpoonup C))_\varphi)/K_{\bowtie} \cong \mu\ (((D \rightharpoonup A \times B \times E) \times (A \rightharpoonup C))_\varphi)_\gamma$$

where

$$\begin{aligned}\gamma(g, h) &\stackrel{\text{def}}{=} \forall x \in dom\, g : \exists y \in dom\, h : \pi_1(g\, x) = y \wedge \\ &\qquad \forall y \in dom\, h : \exists x \in dom\, g : \pi_1(g\, x) = y \\ \mu(g, h) &\stackrel{\text{def}}{=} (g, h)/K_{\bowtie}\end{aligned} \tag{49}$$

Proof: *Immediate.* □

Proposition 4 (BCNF). *Boyce-Codd Normal Form (BCNF) is defined in* SETS *by the reification rules corresponding to 3NF and by the specific reification rule:*

$$(D \rightharpoonup A \times B \times C \times E)_\phi \cong \bowtie ((D \rightharpoonup A \times B \times E) \times (A \rightharpoonup C))_\delta$$

for

$$\delta(g, h) = \varphi(g, h) \wedge \gamma(g, h) \tag{50}$$

Proof: Immediate. □

Example 3 (BCNF). Our toy conference has parallel activities. A parallel activity is identified by a code, has a description and takes place on a date, at a certain hour. Distinct activities with the same description may exist but, in that case, they take place on distinct dates, always at the same hour. At a given hour on a certain date only one parallel activity may start, for operational reasons.

Suppose the starting specification of parallel activities was:

$$PA = (\#Pa \rightharpoonup Description \times Date \times Hour)_{\gamma} \tag{51}$$

where γ has the same definition as ϕ, *cf.* (39). PA is redundant since, for different activities with the same descripion, the information relative to the hour is repeated.

Let us check this redundancy. In PA has two keys, $(\#Pa)$ and $(Date\ Hour)$. Functional dependency

$$Description\ determines\ Hour$$

induces a transitive dependency: the prime attribute $Hour$ is transitively dependant on key $(\#Pa)$. PA is not in BCNF as a consequence of this transitive dependency, although it is in 3NF.

We can convert PA to BCNF by application of Proposition 4. This one cannot be directly applied to PA, which must be transformed into:

$$PA_1 = (\#Pa \rightharpoonup Description \times Date \times Hour \times 1)_{\gamma}$$

where 1 is the identity of $\times$. The outcome will be:

$$PA_2 = ((\#Pa \rightharpoonup Description \times Date) \times (Description \rightharpoonup Hour))_{\delta} \tag{52}$$

where δ corresponds to (50). The table connecting descriptiond to hours eliminates the redundancy.

8.5 Fourth Normal Form

Fourth Normal Form (4NF) is characterized in SETS by a rule which describes the involved reification strategy:

Lemma 11 (4NF characterization). *Fact*

$$(2^{A\times B\times C})_{mvd} \trianglelefteq_{\bowtie} 2^{A\times B} \times 2^{A\times C} \tag{53}$$

holds, where:

$$\begin{aligned} mvd(r) \stackrel{\text{def}}{=} \ & \forall \tau_1, \tau_2 \in r : \pi_1(\tau_1) = \pi_1(\tau_2) \Rightarrow \\ & \exists \tau_3, \tau_4 \in r : (\pi_1(\tau_3) = \pi_1(\tau_4) = \pi_1(\tau_1) \wedge \\ & \qquad \pi_2(\tau_3) = \pi_2(\tau_1) \wedge \\ & \qquad \pi_3(\tau_3) = \pi_3(\tau_2) \wedge \\ & \qquad \pi_2(\tau_4) = \pi_2(\tau_2) \wedge \\ & \qquad \pi_3(\tau_4) = \pi_3(\tau_1)) \end{aligned} \tag{54}$$

$$r \bowtie s \stackrel{\text{def}}{=} \{\langle \pi_1(\tau), \pi_2(\tau), \pi_2(\tau') \rangle | \tau \in r \wedge \tau' \in s \wedge \pi_1(\tau) = \pi_1(\tau')\} \tag{55}$$

Proof: *See [25].* □

So, a table bearing the structure of the antecedent of (53) satisfies the multivalued dependency

$$A\ multidetermines\ B$$

(*cf.* mvd) and A is not a superkey. The table is losslessly decomposed into two tables, in agreement with the multivalued dependencies properties.

Lemma 12 (4NF strict characterization). *4NF is strictly characterized by law:*

$$(2^{A\times B\times C})_{mvd} \cong_\psi (2^{A\times B} \times 2^{A\times C})/K_{\bowtie}$$

where $K_{\bowtie}$ is the kernel relation induced by $\bowtie$ and

$$\psi : (r,s)/K_{\bowtie} \mapsto r \bowtie s$$

Proof: *Immediate.* $\square$

Lemma 13 (4NF concrete domain). *BCNF concrete domain may be defined by either of the following two isomorphic expressions:*

$$(2^{A\times B} \times 2^{A\times C})/K_{\bowtie} \cong \mu\ (2^{A\times B} \times 2^{A\times C})_\gamma$$

where:

$$\begin{aligned}\gamma(r,s) &\stackrel{\text{def}}{=} \forall \tau \in r : \exists \tau' \in s : \pi_1(\tau) = \pi_1(\tau') \wedge \\ &\phantom{\stackrel{\text{def}}{=}} \forall \tau' \in s : \exists \tau \in r : \pi_1(\tau) = \pi_1(\tau') \\ \mu(r,s) &\stackrel{\text{def}}{=} (r,s)/K_{\bowtie}\end{aligned} \tag{56}$$

Proof: *We will prove that μ is a bijection. Firstly, we'll propose the inverse of μ, μ^{-1}:*

$$\begin{aligned}\mu^{-1} : (r,s)/K_{\bowtie} \mapsto\ & (r',s') \\ & \underline{where}\ r' = \{\tau \in r | \exists \tau' \in s : \pi_1(\tau) = \pi_1(\tau')\} \\ & \phantom{\underline{where}\ } s' = \{\tau' \in s | \exists \tau \in r : \pi_1(\tau) = \pi_1(\tau')\}\end{aligned}$$

We have:

$$\begin{aligned}\mu^{-1} \circ \mu((r,s)) &= \mu^{-1}((r,s)/K_{\bowtie}) \\ &= (r',s') \\ &= (\{\tau \in r | TRUE\}, \{\tau' \in s | TRUE\}) \\ &= (r,s)\end{aligned}$$

so μ is injective. We have:

$$\begin{aligned}\mu \circ \mu^{-1}((r,s)/K_{\bowtie}) &= \mu((r',s')) \\ &= (r',s')/K_{\bowtie} \\ &= (r,s)/K_{\bowtie}\end{aligned}$$

so μ is surjective too, as desired.
$\square$

Proposition 5 (4NF). *Fourth Normal Form (4NF) is defined in* SETS *by the reification rules corresponding to BCNF and by the specific reification rule:*

$$(2^{A\times B\times C})_{mvd} \cong \bowtie (2^{A\times B} \times 2^{A\times C})_\gamma \tag{57}$$

Proof: *Immediate.* $\square$

Proposition 5 can be applied in a functional context, after a few transformations. Let us see the general case

$$(C \rightharpoonup A \times B)_\gamma \tag{58}$$

where data invariant

$$\begin{aligned}\gamma(f) \stackrel{\text{def}}{=} \forall x_1, x_2 \in dom\, f : \pi_1(f(x_1)) = \pi_1(f(x_2)) \Rightarrow \\ \exists x_3, x_4 \in dom\, f : \pi_1(f(x_3)) = \pi_1(f(x_4)) = \pi_1(f(x_1)) \wedge \\ \pi_2(f(x_3)) = \pi_2(f(x_1)) \wedge \\ x_3 = x_2 \wedge \\ \pi_2(f(x_4)) = \pi_2(f(x_2)) \wedge \\ x_4 = x_1\end{aligned}$$

enforces multivalued dependency A *multidetermines* B. The data domain affected by data invariant γ, in (58), can be reified according to reification law (8), yielding:

$$C \rightharpoonup A \times B \trianglelefteq^{fdp}_{mkf} 2^{C \times (A \times B)}$$

But we are interested in a similar law attending to the antecedent of (57).

$$C \rightharpoonup A \times B \trianglelefteq^{fdp'}_{mkf'} 2^{A \times B \times C} \tag{59}$$

where mkf' and fdp' follow trivially from mkf and fdp, respectively. Imposing γ to the antecedent of (59) we get the reification step:

$$(C \rightharpoonup A \times B)_\gamma \trianglelefteq^{fdp'}_{mkf'} (2^{A \times B \times C})_{fdp' \wedge \gamma \circ mkf'} \tag{60}$$

by applying Theorem 1. Since

$$\gamma \circ mkf' = mvd$$

assuming fdp' holds, the consequent of (60) can be written as:

$$((2^{A \times B \times C})_{mvd})_{fdp'} \tag{61}$$

So, at last, we are in the able to apply Proposition 5 to (61), as desired.

Example 4 (4NF). Our toy conference problem does not show situations introducing multivalued dependencies, in strict sense, on the existent information. But a functional dependency induces a multivalued dependency, in the general sense.

Let us consider Example 2 again. We saw that, in this situation, redundancy was induced by a functional dependency. We will now consider a reification of the hotels and lodgings records specification (*cf.* (36)), dealing with the functional dependency as a multivalued dependency. We have:

$$HL = (2^{Hn \times (category \times address) \times (\#Part \times di \times Do \times rn)})_\varphi \tag{62}$$

where:

$$\begin{aligned}\varphi(r) \stackrel{\text{def}}{=} mvd(r) \wedge \\ \forall \tau_1, \tau_2 \in r : \pi_1(\pi_3(\tau_1)) = \pi_1(\pi_3(\tau_2)) \Rightarrow \tau_1 = \tau_2\end{aligned}$$

The normalization of HL will now be achieved by application of Proposition 5. This cannot be applied directly since φ has not the desirable form (*i.e.* it unmatches the abstract data invariant of this Proposition). But,

$$\varphi(r) = \varphi_1(r) \wedge \varphi_2(r)$$

where:

$$\varphi_1(r) \stackrel{\text{def}}{=} mvd(r)$$
$$\varphi_2(r) \stackrel{\text{def}}{=} \forall \tau_1, \tau_2 \in r : \pi_1(\pi_3(\tau_1)) = \pi_1(\pi_3(\tau_2)) \Rightarrow \tau_1 = \tau_2$$

Then (62) may be converted into:

$$\begin{aligned} HL_1 &= (HL_1')_{\varphi_2} \\ &\underline{where}\ HL_1' = (2^{Hn \times (category \times address) \times (\#Part \times di \times Do \times rn)})_{\varphi_1} \end{aligned} \tag{63}$$

By applying the above Proposition to HL_1 we obtain:

$$\begin{aligned} HL_2 &= (HL_2')_{\varphi_2'} \\ &\underline{where}\ HL_2' = (2^{Hn \times (category \times address)} \times 2^{Hn \times (\#Part \times di \times Do \times rn)})_{\gamma} \end{aligned} \tag{64}$$

where γ corresponds to (56) and:

$$\varphi_2'(r_1, r_2) \stackrel{\text{def}}{=} \forall \tau_1, \tau_2 \in (r_1 \bowtie r_2) : \pi_1(\pi_3(\tau_1)) = \pi_1(\pi_3(\tau_2)) \Rightarrow \tau_1 = \tau_2 \tag{65}$$

Datatype invariant φ_2 in (63) translates, in (64), to φ_2', which has the same meaning but differs in form. φ_2' is obtained from φ_2 by calculation, following *e.g.* [8, 18]: we start by building the reification diagram of Figure 1, where $\bowtie$ is the abstraction function, *cf.* (55).

$$\begin{tikzcd} HL_1' \arrow[r, "\varphi_2"] & Boole \\ HL_2' \arrow[u, "\bowtie"] \arrow[r, "\varphi_2'"'] & Boole \arrow[u, "id"'] \end{tikzcd}$$

Fig. 1. Reification Diagram of φ_2

This diagram leads to equation [5]:

$$\varphi_2' = \varphi_2 \circ \bowtie \tag{66}$$

φ_2' is the unknown of equation (66) and it is just necessary to calculate its righthand side expression. For $(r_1, r_2) \in HL_2'$, we can apply to it the right hand side of (66):

$$(\varphi_2 \circ \bowtie)(r_1, r_2) \tag{67}$$

From (67), attending to the associativity of function composition, we obtain:

$$\gamma_2(r_1 \bowtie r_2) \tag{68}$$

The *unfolding* of (68) produces:

$$\forall \tau_1, \tau_2 \in (r_1 \bowtie r_2) : \pi_1(\pi_3(\tau_1)) = \pi_1(\pi_3(\tau_2)) \Rightarrow \tau_1 = \tau_2 \tag{69}$$

The expression (69) gives the formal definition of φ_2'. The analysis of (64), when confronted with the results from Example 2, *cf.* (37), leads to the following comments:

- When dealing with a functional dependency as a multivalued dependency we loose the information about the *hotels table* key, in (64).

[5] It should be noted that this equation could have been obtained by applying Theorem 1.

- Datatype invariant φ_2', in conjunction with γ, imposes that $(\#Part)$ is a key for the *lodgings table*, in (64).

In summary, the examples just presented show that:

- 1NF introduces redundancy, the law (16) being an exception.
- The other normal forms do not introduce redundancy. They operate under the scope of $\mathbf{Set}$-isomorphism and just "reshape" the information organization, although with important advantages at several respects, mainly data access and manipulation.

8.6 On Dependence Calculation

The SETS calculus is particularly useful at dependency calculation level. It makes for the stepwise introduction of such dependencies throughout a reification reasoning process, for instance, in the context of the application of Proposition 5.

Functional dependencies are easy to calculate in SETS — recall (8), which expresses a basic interconnection between mappings and binary relations: any mapping is a particular case of a relation. Put in another way: every relation bearing a functional dependence (fdp) can be converted (via mkf) into a finite mapping.

Multivalued dependencies do not generate an as simple interconnection between two mathematical objects. Consider a relation with relational scheme $R = A\ B\ C$ satisfying multivalued dependency $A \twoheadrightarrow B$. Then, to each value in A corresponds a set of values in B and another set of values in C. The relation contains all the possible combinations of values from these two sets, corresponding to the value in A. The transposition of this informal concept to SETS leads, in a first approach, to calculation law:

$$A \rightharpoonup (2^{B\times C})_\mu \trianglelefteq_{\alpha}^{mvdp'} 2^A \times 2^{A\times B\times C} \tag{70}$$

where

$$\mu(r) \stackrel{\mathrm{def}}{=} r = dom\, r \times rng(r) \tag{71}$$

$$mvdp'(\_, r) \stackrel{\mathrm{def}}{=} mvd(r) \tag{72}$$

$$\alpha(c, r) \stackrel{\mathrm{def}}{=} [a \mapsto \emptyset | a \in c] \dagger [a \mapsto \{\langle x, y\rangle | \langle a, x, y\rangle \in r\} | \langle a, b, c\rangle \in r] \tag{73}$$

Fact (70) admits an equivalent formulation:

$$A \rightharpoonup (B \rightharpoonup 2_+^C)_{\mu'} \trianglelefteq_{\alpha'}^{mvdp'} 2^A \times 2^{A\times B\times C} \tag{74}$$

where

$$\mu'(f) \stackrel{\mathrm{def}}{=} \forall b_1, b_2 \in dom\, f : f(b_1) = f(b_2) \tag{75}$$

$$\alpha'(c, r) \stackrel{\mathrm{def}}{=} [a \mapsto [\,] | a \in c] \dagger [a \mapsto [x \mapsto \{y | \langle a, x, y\rangle \in r\} | \langle a, x, c\rangle \in r] | \langle a, b, c\rangle \in r] \tag{76}$$

Fact (74) can be generalized, by replacing 2^C for 2_+^C and originating the expected result:

Proposition 6. *Multivalued dependencies can be calculated in* SETS *via the reification rule which follows.*

$$(A \rightharpoonup (B \rightharpoonup (C \rightharpoonup 1)))_\phi \trianglelefteq_{mknf}^{mvdp} 2^A \times 2^{A\times B} \times 2^{A\times B\times C} \tag{77}$$

where

$$\phi(f) \stackrel{\mathrm{def}}{=} \forall x \in dom\, f : \forall y, y' \in dom(f\, x) : f(x)(y) = f(x)(y') \tag{78}$$

$$mvdp(\_, \_, r) \stackrel{\mathrm{def}}{=} mvd(r)$$

$$\begin{aligned} mknf(c_1, c_2, r) \stackrel{\mathrm{def}}{=} &\ [a \mapsto \bot | a \in c_1] \dagger \\ &\ [a \mapsto [b \mapsto \bot | \langle a, b\rangle \in c_2] | \langle a, b\rangle \in c_2] \dagger \\ &\ [a \mapsto [b \mapsto [c \mapsto \top | \langle a, b, c\rangle \in r] | \langle a, b, c\rangle \in r] | \langle a, b, c\rangle \in r] \end{aligned}$$

The abstract domain of (77) is complex. This corroborates the intuition that multivalued dependencies which do not follow from functional dependencies, do not emerge easily from the formal specification of data models. Finally, note that the chained application of rule (21) would lead us anyway to the basis of (77):

$$(A \rightharpoonup (B \rightharpoonup (C \rightharpoonup 1))) \trianglelefteq 2^{A} \times 2^{A \times B} \times 2^{A \times B \times C} \tag{79}$$

cf. reification sequence:

$$\begin{aligned} A \rightharpoonup (B \rightharpoonup (C \rightharpoonup 1)) &\trianglelefteq 2^{A} \times ((A \times B) \rightharpoonup (C \rightharpoonup 1)) \\ &\trianglelefteq 2^{A} \times 2^{A \times B} \times ((A \times B \times C) \rightharpoonup 1) \\ &\cong 2^{A} \times 2^{A \times B} \times 2^{A \times B \times C} \end{aligned}$$

9 Conclusions and Future Work

The research carried out in this report belongs to the intersection of formal methods with relational database theory. On the one hand, SETS, a data reification calculus, was used to formally re-interpret standard normalization-theory. On the other hand, the set of calculation rules available from the literature was enriched with further derived laws and was made more systematic. In particular, a calculation rule for multivalued dependencies was presented which complements an already existing rule for functional dependency calculation.

One of the first applications of SETS to industrial relational data base design is reported in [15]. The fact that a normalized database was obtained was regarded as natural, but there was no clear understanding of the relation between data *calculation* and *normalization*.

The main contribution of this report (which builds upon the master's thesis [25] of one of the authors) is to identify the class of SETS calculation laws which express what is called *data normalization* in relational database theory. Most of the relational database calculations reported so far in the SETS literature (see [26] for another industrial design) belong to 1NF, bringing 3NF and even 4NF for free (as there are no explicit multivalued dependencies), recall Proposition 1 — laws which remove structure from data domain specifications by redundancy introduction.

We have realized that, besides these laws, there is a set of laws (*cf.* Proposition 2 to 5) which solve violation of certain normal forms, caused by partial dependencies, transitive dependencies and multivalued dependencies. These are isomorphism laws which eliminate a "semantic" datatype invariant and induce a "concrete" data invariant in the reification process corresponding to the normalization task taking place. They are applicable to data domain specifications satisfying some properties (expressed in the semantic data invariant) typical of databases — functional dependencies and multivalued dependencies. The resulting data domains are the classes of all pairs of "tables" where all avoidable "entries" are eliminated (via the concrete datatype invariant).

Previously, the laws belonging to 1NF were enough to garantee normalization since the specifications were not enriched by the properties referred to above. (Maybe there was lack of sensibility to the dependencies or else they were considered "minor" properties.)

In the end, then, what is the relationship between reification and normalization? In a normalization process one table is decomposed into many tables, the objective being to eliminate information redundancy. Underlying is the *lossless decomposition* concept. Thus, every normalization process corresponds to the *implementation* of one given table by various tables. The join function is the abstraction function. This implementation is rigorous wrt. all normal forms, except the 1NF. With 1NF the normalization process just eliminates structure (attributes with structured domain), decomposing a table into two or more tables. However, the properties exhibited by the resulting tables are not recorded.

Therefore, classical normalization theory may be considered as a reification theory as understood in SETS, up to 1NF fault. We now ask what is the relation between SETS and the Relational Calculus (consisting of the normalization theory without the generalizations of 4NF). Our answer is:

the Relational Calculus is a SETS *subcalculus*

in the sense that any law of the Relational Calculus (*i.e.* a normalization step) can be interpreted in SETS as a reification law, *cf.* Propositions 1 to 5. On the other side, SETS contains laws involving recursive data domains (see [20, 21]) which have no translation in the relational calculus. Moreover, a fundamental handicap of the relational calculus is the lack of a constructive approach to data design, which in **Set** has a categorial, *functorial* basis — recall [21] again.

The SETS approach to formal software development has been regularly taught at the **University of Minho** since 1988. It has become influential in the way **database design** is taught to Minho's undergraduate students. These are stimulated to obatin normalized data models from (possibly recursive) formal data domain specifications by calculation, in the style presented in this report. The approach includes a SETS formalization of E-R diagrams [25]. Many students have reported the successful application of the calculus at industrial software design level.

The fact that SETS crosses the boundaries of different programming paradigms (*e.g.* C programming, functional programming, relational database programming) makes it very easy to obtain a programming media independent design method. This paves the way to formal reverse engineering, as has been shown in [24], [17] and [29].

References

1. CODASYL. CODASYL Data Base Task Group April 71 Report, Apr. 1971. ACM, New York.
2. E. F. Codd. A relational model of data for large shared data banks. *CACM*, 13(6):377–387, June 1970.
3. E. F. Codd. A database sublanguage founded on the relational calculus. In *ACM SIGFIDET Workshop on Data Description, Access and Control*, pages 35–61, Nov. 1971.
4. E. F. Codd. Normalized database structure: A brief tutorial. In *ACM SIGFIDET Workshop on Data Description, Access and Control*, pages 1–17, Nov. 1971.
5. E. F. Codd. Further normalization of the database relational model. In *Data Base Systems*, pages 33–64. Prentice-Hall, 1972. Courant Inst. Computer Science Symp. 6, Englewood Cliffs, NJ.
6. E. F. Codd. Relational completeness of database sublanguages. In *Data Base Systems*, pages 65–98. Prentice-Hall, 1972. Courant Inst. Computer Science Symp. 6, Englewood Cliffs, NJ.
7. E. F. Codd. Recent investigations in relational database systems. In *1974 IFIP Conf.*, pages 1017–1021, 1974.
8. J. Darlington. Program transformation. In *Funct. Prog. and Its Applications: An Advanced Course*. Cambridge Univ. Press, 1982.
9. C. B. Jones. *Software Development — A Rigorous Approach*. Series in Computer Science. Prentice-Hall International, 1980. C. A. R. Hoare.
10. C. B. Jones. *Systematic Software Development Using* VDM. Series in Computer Science. Prentice-Hall International, 1986. C. A. R. Hoare.
11. I. S. Jourdan. Reificação de tipos abstractos de dados: Uma abordagem matemática. Master's thesis, University of Coimbra, 1992. (In Portuguese).
12. D. Maier. *The Theory of Relational Databases*. Computer Science Press, 1983.
13. G. Malcolm. Data structures and program transformation. *Science of Computer Programming*, 14:255–279, 1990.
14. J. Martin. *Information Engineering*, volume 1. Savant Research Studies, Aug. 1986.
15. C. Moreira, A. Reis, R. Jesus, and A. Barros. Especificação formal de um sistema para conservatórias de registo predial. Technical Report CCES-OP3:R1/88, U. Minho, May 1988.
16. C. Morgan. *Programming from Specification*. Series in Computer Science. Prentice-Hall International, 1990. C. A. R. Hoare, series editor.
17. F. L. Neves and J. N. Oliveira. Software Reuse by Model Reification . In WISR'95 - 6th Annual Workshop on Software Reuse, August 28–30 1995. Charles Il, Illinois, USA.
18. J. N. Oliveira. The transformational paradigm as a means of smoothing abrupt softwar e design steps. Technical Report CCES-JNO:R2/85, U. Minho, December 1985.
19. J. N. Oliveira. Refinamento transformacional de especificações (terminais). In *Actas das XII "Jornadas Luso-Espanholas de Matemática"*, volume II, pages 412–417, Braga, Portugal, 4-8 May 1987.
20. J. N. Oliveira. *A Reification Calculus for Model-Oriented Software Specification* . Formal Aspects of Computing, 2(1):1–23, April 1990.
21. J. N. Oliveira. *Software Reification using the SETS Calculus* . In *Proc. of the BCS FACS 5th Refinement Workshop, Theory and Practice of Formal Software Development, London, UK*, pages 140–171. Springer-Verlag, 8–10 January 1992. (Invited paper).
22. J. N. Oliveira. *Métodos Formais de Programação*. University of Minho, 4^{th} edition, 1997. Textbook (489 p. in Portuguese [6]). English version under preparation at the time of writing.

[6] 846K gzipped PS file available from `http://www.di.uminho.pt/~jno/ps/mfp.ps.gz`.

23. J. N. Oliveira. SETS — a data structuring calculus and its application to program development. Technical report, *UNU/IIST*, May 1997. Lecture Notes of Course at *UNU/IIST*, Macau, 5-16 May 1997, 125p. [7].
24. J. N. Oliveira and A. M. Cruz. Formal calculi applied to software component knowledge elicitation. Technical Report C19-WP2D, DI/INESC, December 1993. IMI Contract *C.1.9. Sviluppo di Metodologie, Sistemi e Servizi Innovativi in Rete.* [8].
25. C. J. Rodrigues. Sobre o desenvolvimento formal de bases de dados. Master's thesis, University of Minho, 1993. (In Portuguese).
26. N. Rodrigues. Implementação de bases de dados em ORACLE. Technical report, SONAE & U. Minho, 1991. Workstudy in industry.
27. J. M. Spivey. *The Z Notation — A Reference Manual*. Series in Computer Science. Prentice-Hall International, 1989. C. A. R. Hoare.
28. J. Ullman. *Principles of Database Systems*. Computer Science Press, 1982.
29. M.J.P. Veloso. Engenharia reversa em larga escala de sistema de informação — caso concreto: O sistema de informação dos S.A. Technical Report LESI-Op-IIIb, Universidade do Minho, Fev. 1995. (In Portuguese).

A Some Primitive Functions and Predicates

$$\langle f, h\rangle\, a \stackrel{\text{def}}{=} (f\, a, h\, a) \tag{80}$$

$$\underline{c_0} \stackrel{\text{def}}{=} \lambda x.c_0 \tag{81}$$

$$collect(\rho) \stackrel{\text{def}}{=} \begin{pmatrix} a \\ \{x \in B \mid\ a\rho x\} \end{pmatrix}_{a \in 2^{\pi_1}(\rho)} \tag{82}$$

$$dpi(\sigma, \tau) \stackrel{\text{def}}{=} 2^{\pi_1}(dom\, \tau) \subseteq dom\, \sigma \tag{83}$$

$$eqd((\sigma, \tau)) \stackrel{\text{def}}{=} dom\, \sigma = dom\, \tau \tag{84}$$

$$elems(b) \stackrel{\text{def}}{=} \{b(i) \mid\ i \in length(b)\} \tag{85}$$

$$fdp(r) \stackrel{\text{def}}{=} \forall t, t' \in r : \pi_1(t) = \pi_1(t') \Rightarrow \pi_2(t) = \pi_2(t') \tag{86}$$

$$\sigma \bowtie \tau \stackrel{\text{def}}{=} \begin{pmatrix} a \\ (\sigma(a), \tau(a)) \end{pmatrix}_{a \in dom\, \sigma} \tag{87}$$

$$set2fm(r) \stackrel{\text{def}}{=} \begin{pmatrix} p \\ NIL \end{pmatrix}_{p \in r} \tag{88}$$

$$\bowtie_{\pi} \stackrel{\text{def}}{=} \bowtie \circ \langle \pi_1, \dagger \circ ((A \rightharpoonup \underline{(\,)}) \times pcurry)\rangle \tag{89}$$

$$mkf(\rho) \stackrel{\text{def}}{=} \begin{pmatrix} \pi_1(t) \\ \pi_2(t) \end{pmatrix}_{t \in \rho} \tag{90}$$

$$pcurry(f) \stackrel{\text{def}}{=} \begin{pmatrix} a \\ \begin{pmatrix} \pi_2(p) \\ f(p) \end{pmatrix}_{p \in dom\, f \wedge \pi_1(p) = a} \end{pmatrix}_{a \in 2^{\pi_1}(dom\, f)} \tag{91}$$

[7] 305K .ps.gz file available from `http://www.di.uminho.pt/~jno/ps/unu97set.ps.gz`.
[8] 167K ps.gz file available from `http://www.di.uminho.pt/~jno/ps/c1-wp2b.ps.gz`.