Journal of Logical and Algebraic Methods in Programming 83 (2014) 249–262

Contents lists available at ScienceDirect

Journal of Logical and Algebraic Methods in Programming

www.elsevier.com/locate/jlamp

A relation-algebraic approach to the "Hoare logic" of functional dependencies

José N. Oliveira [a]

[a] *High Assurance Software Lab, INESC TEC and Univ. Minho, 4710-070 Braga, Portugal*

ARTICLE INFO

Article history:
Available online 18 February 2014

Keywords:
Relational mathematics
Program calculi
Data dependency theory

ABSTRACT

Abstract algebra has the power to *unify* seemingly disparate theories once they are encoded into the same abstract formalism. This paper shows how a relation-algebraic rendering of both database dependency theory and Hoare programming logic purports one such unification, in spite of the latter being an algorithmic theory and the former a data theory. The approach equips relational data with *functional types* and an associated type system which is useful for database operation type checking and optimization.
The prospect of a generic, unified approach to both programming and data theories on top of libraries already available in automated deduction systems is envisaged.

© 2014 Elsevier Inc. All rights reserved.

"*Hardly anybody confronted with practical problems knows how to apply relational calculi* [*for which*] *there is almost no broadly available computer support* [...] *We feel, however, that the situation is about to change dramatically as relational mathematics develops and computer power exceeds previous expectations.*"

[Gunther Schmidt [1]]

1. Introduction

In a paper addressing the influence of Alfred Tarski (1901–1983) in computer science, Solomon Feferman [2] quotes the following statement by his colleague John Etchemendy: "*You see those big shiny Oracle towers on Highway 101? They would never have been built without Tarski's work on the recursive definitions of satisfaction and truth*".

The 'big shiny Oracle towers' are nothing but the headquarters of Oracle Corporation, the giant database software provider sited in the San Francisco Peninsula. Still Feferman [2]: "*Does Larry Ellison know who Tarski is or anything about his work?* [...] *I learned subsequently from Jan Van den Bussche that* [...] *he marks the reading of Codd's seminal paper as the starting point leading to the Oracle Corporation.*"

Bussche [3] had in fact devoted attention to relating Codd and Tarski's work: "*We conclude that Tarski produced two alternatives for Codd's relational algebra: cylindric set algebra, and relational algebra with pairing* [...] *For example, we can represent the ternary relation* $\{(a, b, c), (d, e, f)\}$ *as* $\{(a, (b, c)), (d, (e, f))\}$". Still Bussche [3]:

"*Using such representations, we leave it as an exercise to the reader to simulate Codd's relational algebra in* RA^{+} [relational algebra with pairing]".

E-mail address: jno@di.uminho.pt.

http://dx.doi.org/10.1016/j.jlap.2014.02.013
2352-2208/© 2014 Elsevier Inc. All rights reserved.

To the best of the author's knowledge, nobody has thus far addressed this *exercise* in a thorough and generic way. Instead, standard relational database theory [4,5] includes a well-known relation algebra but this is worked out in set theory and quantified logic, far from the objectives of Tarski's life-long pursuit in developing methods for elimination of quantifiers from logic expressions. This effort ultimately lead to his *formalization of set theory without variables* [6].

The topic has acquired recent interest with the advent of work on implementing extensions of Tarski's algebra in automated deduction systems such as *Isabelle* [7] or *Prover9* and the associated counterexample generator *Mace4* [8]. This offers a potential for automation which has not been acknowledged by the database community. In this context, it is worth mentioning an early concern of the founding fathers of the standard theory [9]:

"[*A*] *general theory that ties together dependencies, relations and operations on relations is still lacking*".

More than 30 years later, this concern is still justified, as database programming standards remain insensitive to techniques such as formal verification and extended static checking [10] which are regarded more and more essential to ensuring quality in complex software systems.

Contribution The remainder of this paper will show how an algebraic treatment of standard data dependency theory along the *exercise* proposed by Bussche equips relational data with *functional types* and an associated type system which can be used to *type check* database operations.

Interestingly, such a *typed* approach to database programming will be shown to relate to other programming logics such as e.g. *Hoare logic* [11] or that of *strongest invariant functions* [12] which has been used in the analysis of while statements, for instance.

On the whole, the approach has a *unifying theories of programming* [13] flavour, even though the exercise will not be carried out in "canonical" UTP.

Outline Section 2 introduces functional dependencies (FD) and shows how to convert the standard definition into the Tarskian, quantifier-free style. The parallel between the *functions as types* approach which emerges from such a conversion and a similar treatment of Hoare logic starts in Section 3. Section 4 shows that, in essence, *injectivity* is what matters in FDs and gives a correspondingly simpler definition of FD which is used in Section 5 to re-factor the standard theory into a *type system of FDs*. Section 6 shows how to use this type system to type check database operations and Section 7 shows how to calculate query optimizations from FDs. The last sections conclude and give an account of related and future work.

Some technical details are omitted from the current paper for conciseness. All can be found in a technical report available on-line [14].

2. Introducing functional dependencies

In standard relational data processing, real life objects or entities are recorded by assigning values to their observable properties or *attributes*. A database *table* is a collection of such attribute assignments, one per object, such that all values of a particular attribute (say i) are of the same type (say A_i). For n such attributes, a *relational database file* T can be regarded as a set of n-tuples, that is, $T \subseteq A_1 \times \cdots \times A_n$. A *relational database* is just a collection of several such relations, or tables.

Attribute names normally replace natural numbers in the identification of attributes. The enumeration of all attribute names in a database table, for instance $S = \{\text{PILOT}, \text{FLIGHT}, \text{DATE}, \text{DEPARTS}\}$ concerning the airline scheduling system given as example in [4], is a finite set called the table's *scheme*. This scheme captures the *syntax* of the data. What about *semantics*? Even non-experts in airline scheduling will accept "business rules" such as, for instance: *a single pilot is assigned to a given flight, on a given date*. This restriction is an example of a so-called *functional dependency* (FD) among attributes, which can be stated more formally by writing "FLIGHT DATE $\rightarrow$ PILOT" to mean that *attribute* PILOT *is functionally dependent on* FLIGHT *and* DATE, or that FLIGHT, DATE *functionally determine* PILOT.

Data dependencies help in capturing the *meaning* of relational data. Data dependency theory involves not only functional dependencies (FD) but also multi-valued dependencies (MVD). Both are central to the standard theory, where they are addressed in an axiomatic way. Maier [4] provides the following definition for FD-satisfiability:

Definition 1. Given subsets $x, y \subseteq S$ of the relation scheme S of an n-ary relation T, this relation is said to satisfy functional dependency $x \rightarrow y$ iff all pairs of tuples $t, t' \in T$ which "agree" on x also "agree" on y, that is,

$$\forall t, t' : t, t' \in T \quad \Rightarrow \quad \big(t[x] = t'[x] \quad \Rightarrow \quad t[y] = t'[y]\big) \tag{1}$$

(The notation $t[x]$ in (1) means "the values exhibited by tuple t for the attributes in x".) □

How does one express formula (1) in Tarski's relation algebra style, without the two-dimensional universal quantification and logical implications inside? For so doing we need to settle some notation. To begin with, $t[x]$ is better written as $x(t)$, where x is identified with the *projection function* associated to attribute set x. Regarding x and y in (1) as such functions we write:

$$\forall t, t' : t, t' \in T \quad \Rightarrow \quad \big(x(t) = x(t') \quad \Rightarrow \quad y(t) = y(t')\big) \tag{2}$$

Next, we observe that, given a function $f : A \to B$, the binary relation $R \subseteq A \times A$ which checks whether two values of A have the same image under f[1] — that is, $a' R a \equiv f(a') = f(a)$ — can be written alternatively as $a'(f^\circ \cdot f)a$. Here, f° denotes the *converse* of f — that is, $a(f^\circ)b$ holds iff $b = f(a)$ — and the dot $(\cdot)$ denotes the extension of function composition to binary relations[2]:

$$b(R \cdot S)c \quad \equiv \quad \exists a : b \; R \; a \; \wedge \; a \; S \; c \tag{3}$$

Using converse and composition the rightmost implication of (2) can be rewritten into $t(x^\circ \cdot x)t' \Rightarrow t(y^\circ \cdot y)t'$, for all $t, t' \in T$. Implications such as this can be expressed as relation inclusions, following the definition

$$R \subseteq S \quad \equiv \quad \forall b, a : b \; R \; a \quad \Rightarrow \quad b \; S \; a \tag{4}$$

However, just stating the inclusion $x^\circ \cdot x \subseteq y^\circ \cdot y$ would be a gross error, for the double scope of the quantification $(t \in T \; \wedge \; t' \in T)$ would not be taken into account. To handle this, we first unnest the two implications of (2),

$$\forall t, t' : \big(t \in T \; \wedge \; t' \in T \; \wedge \; t(x^\circ \cdot x)t'\big) \quad \Rightarrow \quad t(y^\circ \cdot y)t'$$

and treat the antecedent $t \in T \; \wedge \; t' \in T \; \wedge \; t(x^\circ \cdot x)t'$ independently, by replacing the set of tuples T by the binary relation $[\![T]\!]$ defined as follows[3]:

$$b[\![T]\!]a \quad \equiv \quad b = a \; \wedge \; a \in T \tag{5}$$

Note that $t \in T$ can be expressed in terms of $[\![T]\!]$ by $\exists u : t[\![T]\!]u$ and similarly for $t' \in T$. Then:

$$\begin{array}{ll} & \big(t \in T \; \wedge \; t' \in T \; \wedge \; t(x^\circ \cdot x)t'\big) \\ \equiv & \quad \{\text{expansion of } t \in T \text{ and } t' \in T\} \\ & \exists u, u' : t[\![T]\!]u \; \wedge \; t'[\![T]\!]u' \; \wedge \; t(x^\circ \cdot x)t' \\ \equiv & \quad \{\wedge \text{ is commutative; } u = t \text{ and } u' = t'; \text{converse}\} \\ & \exists u, u' : t[\![T]\!]u \; \wedge \; u(x^\circ \cdot x)u' \; \wedge \; u'[\![T]\!]^\circ t' \\ \equiv & \quad \{\text{composition (3) twice}\} \\ & t\big([\![T]\!] \cdot x^\circ \cdot x \cdot [\![T]\!]^\circ\big)t' \qquad \square \end{array}$$

Finally, by putting this together with $t(y^\circ \cdot y)t'$ we obtain

$$[\![T]\!] \cdot x^\circ \cdot x \cdot [\![T]\!]^\circ \subseteq y^\circ \cdot y \tag{6}$$

as a quantifier-free relation algebra expression meaning the same as (1).

Generalization To reassure the reader worried about the doubtful practicality of derivations such as the above, we would like to say that we don't need to do it over and over again: inequality (6), our Tarskian alternative to the original textbook definition (1), is all we need for calculating with functional dependencies. Moreover, we can start this by actually expanding the scope of the definition from sets of tuples $[\![T]\!]$ and attribute functions (x, y) to arbitrary binary relations R and suitably typed functions f and g:

$$R \cdot f^\circ \cdot f \cdot R^\circ \subseteq g^\circ \cdot g \tag{7}$$

In this wider setting, R can be regarded not only as a piece of data but also as the specification of a non-deterministic computation, or even the transition relation of a finite-state automaton; and f (resp. g) as a function which observes the input (resp. output) of R. Put back into quantified logic, such a wider notion of a functional dependency will expand as follows:

$$\forall a', a : f(a') = f(a) \quad \Rightarrow \quad \big(\forall b', b : b' \; R \; a' \; \wedge \; b \; R \; a \quad \Rightarrow \quad g(b') = g(b)\big) \tag{8}$$

[1] This is known as the *nucleus* [12] or *kernel* [15] of a function f.

[2] Thus composition of both functions are relations should be read backwards. This is consistent with $b \; f \; a$ (function f regarded as a special case of relation) meaning $b = f(a)$ and not $a = f(b)$.

[3] This is a standard way of encoding a set T as a *partial identity* [1], thus called since $[\![T]\!] \subseteq id$. The set of all such relations forms a Boolean algebra which reproduces the usual algebra of sets. Moreover, partial identities are symmetric ($[\![T]\!]^\circ = [\![T]\!]$) and such that $[\![S]\!] \cdot [\![T]\!] = [\![S]\!] \cap [\![T]\!]$. Also known as *coreflexives* [16] or as *monotypes* [17], partial identities are special cases of *tests* in Kleene algebras with tests [18].

In words: *inputs a, a' to R which are indistinguishable by f can only lead to outputs indistinguishable by g*. Notationally, we will convey this interpretation by writing $R : f \to g$ or $f \xrightarrow{R} g$. We can still say that R satisfies FD $f \to g$, in particular wherever R is a piece of data. As can be easily checked, $f(a') = f(a)$ is an equivalence relation which, in the wider setting, can be regarded as the *semantics* of the datatype which R takes inputs from (think of $f : A \to B$ as a *semantic* function mapping a syntactic domain A into a semantic domain B), and similarly for g concerning the output type.

Summing up, the functions f and g in (7) can be regarded as *types* for R. Some type assertions of this kind will be very easy to check, for instance $id : f \to f$, just by replacing $R, f, g := id, f, f$ in (7) and simplifying. But type inference will be easier to calculate on top of the even simpler (re)statement of (7) which is given next.

3. Functions as types

Before proceeding let us record two properties of the relational operators *converse* and *composition*[4]:

$$(R \cdot S)^\circ = S^\circ \cdot R^\circ \tag{9}$$

$$\left(R^\circ\right)^\circ = R \tag{10}$$

Moreover, it will be convenient to have a name for the relation $R^\circ \cdot R$ which, for R a function f, is the equivalence relation "indistinguishable by f" seen above. We define

$$\ker R \triangleq R^\circ \cdot R \tag{11}$$

and read $\ker R$ as "the *kernel* of R". Clearly, $a'(\ker R)a$ means $\exists b : b\ R\ a' \wedge b\ R\ a$ and therefore $\ker R$ measures the *injectivity* of R: the larger it is the larger the set of inputs which R is unable to distinguish (= the less *injective* R is).

We capture this by introducing a preorder on relations which compares their *injectivity*:

$$R \leqslant S \triangleq \ker S \subseteq \ker R \tag{12}$$

As an example, take two list functions: *elems* computing the set of all elements of a list and *bagify* keeping the bag of such elements. The former loses more information (order and multiplicity) than the latter, which only forgets about order. Thus $elems \leqslant bagify$. A function f (relation in general) will be *injective* iff $\ker f \subseteq id$ ($id \leqslant f$), which easily converts to the usual definition: $f(a') = f(a) \Rightarrow a' = a$.

Summing up: for functions or any totally defined relations R and S,[5] $R \leqslant S$ means that *R is less or as injective as S*; for possibly partial R and S, it will mean that *R less injective or more defined than S*. Therefore, for *total* relations R the preorder is universally bounded, $! \leqslant R \leqslant id$, where the infimum is captured by constant function ! which maps every argument to a given (predefined) value, the choice of which is irrelevant.[6] The kernel of ! is therefore the largest possible, denoted by $\top$ (for "top"). The other bound is trivial to check, since $\ker id = id$, this arising from the well-known fact that id is the unit of composition. In general, $id \leqslant R$ means that R is injective.

Equipped with this ordering, we may spruce up our relational characterization of the $f \xrightarrow{R} g$ type assertion, or functional dependency (FD):

$$\begin{array}{ll} & f \xrightarrow{R} g \\ \equiv & \quad \{\text{definition (7)}\} \\ & R \cdot f^\circ \cdot f \cdot R^\circ \subseteq g^\circ \cdot g \\ \equiv & \quad \{\text{converses (9), (10); kernel (11)}\} \\ & \ker(f \cdot R^\circ) \subseteq \ker g \\ \equiv & \quad \{(12)\text{: } g \text{ is "}\textit{less or as injective}\text{ as } f \text{ w.r.t. } R\text{"}\} \\ & g \leqslant f \cdot R^\circ \qquad \square \end{array}$$

We thus reach a rather compact formula for expressing functional dependencies, whose layout invites us to actually swap the direction of the arrow notation (but, of course, this is optional and just a matter of taste):

[4] It may help to recall the same properties from elementary linear algebra, once converse is interpreted as *matrix transposition* and composition as *matrix–matrix multiplication* [1].

[5] A relation R is totally defined (or *entire*) iff $id \subseteq \ker R$.

[6] Thus $! \cdot f = !$, for all f. Also note that $R \leqslant S$ is a preorder, not a partial order, meaning that two relations indistinguishable with respect to their degree of injectivity can be different.

Definition 2. Given an arbitrary binary relation $R \subseteq A \times B$ and functions $f : B \to D$ and $g : A \to C$, the "type assertion" $g \xleftarrow{R} f$ meaning that R *satisfies FD* $f \to g$ is captured by the equivalence:

$$g \xleftarrow{R} f \equiv g \leqslant f \cdot R^\circ \qquad \square \tag{13}$$

There are two main advantages in definition (13), besides saving ink. The most important is that it profits from the relational calculus of injectivity which will be addressed in the following section. The other is that it makes it easy to bridge with other programming logics, as is seen next.

Parallel with Hoare logic As is widely known, Hoare logic is based on triples of the form $\{p\}R\{q\}$, with the standard interpretation: "*if the assertion p is true before initiation of a program R, then the assertion q will be true on its completion*" [11].

Let program R be identified with the relation which captures its state transition semantics and predicates p (and q) be identified with relation $s'[\![p]\!]s \equiv s' = s \ \wedge \ p(s)$ (similarly for q) in which the reader identifies the earlier trick of converting sets to partial identities (Section 2). Note how $[\![p]\!]$ can be regarded as the semantics of a statement which checks $p(s)$ and does not change state, failing otherwise. In relation algebra the Hoare triple is captured by

$$\{p\}R\{q\} \quad \equiv \quad \mathsf{rng}\left(R \cdot [\![p]\!]\right) \subseteq [\![q]\!] \tag{14}$$

meaning that the outputs of R (given by the range operator rng) for inputs pre-conditioned by p fall inside q[7]; that is, q is *weaker* than the strongest (*liberal*) post-condition $slp(R, p)$, something we can express by writing

$$\{p\}R\{q\} \quad \equiv \quad q \leqslant p \cdot R^\circ \tag{15}$$

under a suitable preorder $\leqslant$ expressing that q is less constrained than $p \cdot R^\circ$[8]:

$$R \leqslant S \quad \equiv \quad \mathsf{dom}\, S \subseteq \mathsf{dom}\, R \tag{16}$$

In spite of the different semantic context, there is a striking formal similarity between formulas (15) and (13) suggesting that Hoare logic and the logic we want to build for FDs share the same mathematics once expressed in relation algebra. Such similarities will become apparent in the sequel, particularly whereupon we write $p \xrightarrow{R} q$ (or the equivalent $q \xleftarrow{R} p$) for $\{p\}R\{q\}$ to put the two notations closer to each other. In this way, rules such as e.g. that of composition, $\{p\}R\{q\} \ \wedge \ \{q\}S\{r\} \Rightarrow \{p\}R; S\{r\}$ become reminiscent of labeled transition systems[9]:

$$p \xrightarrow{R} q \ \wedge \ q \xrightarrow{S} r \quad \Rightarrow \quad p \xrightarrow{R;S} r \tag{17}$$

We will check the FD equivalent to composition rule (17) shortly.

4. A calculus of injectivity ($\leqslant$)

One of the advantages of relation algebra is its easy "tuning" to special needs, which we will illustrate below concerning the algebra of injectivity. We give just an example, taken from [14]; the reader is referred to this report for technical details.

We start by considering two rules of relation algebra which prove very useful in program calculation:

$$f \cdot R \subseteq S \quad \equiv \quad R \subseteq f^\circ \cdot S \tag{18}$$

$$R \cdot f^\circ \subseteq S \quad \equiv \quad R \subseteq S \cdot f \tag{19}$$

In these equivalences,[10] which are widely known as *shunting rules* [23,1], f is required to be a (total) function. In essence, they let one trade a function f from one side to the other of a $\subseteq$-equation just by taking converses. (This is akin to "changing sign" in trading terms in inequations of elementary algebra.)

It would be useful to have similar rules for the injectivity preorder, which we have chosen as support for our definition of a FD (13). Such rules turn out to be quite easy to infer, as is the case of the following Galois connection for trading a function f with respect to injectivity

$$R \cdot f \leqslant S \quad \equiv \quad R \leqslant S \cdot f^\circ \tag{20}$$

[7] See e.g. [19]. Term $\mathsf{rng}(R \cdot [\![p]\!])$ instantiates the semiring *diamond* combinator of [20]. Wehrman et al. [21] give an even simpler semantics for Hoare triples: $P\{R\}Q \equiv R \cdot P \subseteq Q$, that is, P is at most the *weakest pre-specification* (residual relation) $R \setminus Q$, where $b(R \setminus Q)a$ means $\forall c : c \ R \ b \Rightarrow c \ Q \ a$ [22].

[8] Details: by definition, $\mathsf{dom}\, R = \ker R \cap id$ and $\mathsf{rng}\, R = \mathsf{dom}\,(R^\circ)$ (converse duality). Starting from (14), triple $\{p\}R\{q\}$ asserts $\mathsf{rng}\,(R \cdot [\![p]\!]) \subseteq [\![q]\!]$, itself the same as $\mathsf{dom}\,([\![p]\!] \cdot R^\circ) \subseteq \mathsf{dom}\,[\![q]\!]$ (15) by converse duality and the fact that the domain of a partial identity is itself. Parentheses $[\![\_]\!]$ are dropped for improved readability.

[9] Forward composition $R; S$ means the same as $S \cdot R$.

[10] Technically, these equivalences should be regarded as (families of) Galois connections.

calculated as follows:

$$
\begin{array}{ll}
& R \cdot f \leqslant S \\
\equiv & \quad \{\text{definition (12); converses (9), (10); kernel (11)}\} \\
& \ker S \subseteq f^\circ \cdot (\ker R) \cdot f \\
\equiv & \quad \{\text{shunting rules (18), (19)}\} \\
& f \cdot \ker S \cdot f^\circ \subseteq \ker R \\
\equiv & \quad \{\text{converses, kernel and definition (12) again}\} \\
& R \leqslant S \cdot f^\circ \qquad \square
\end{array}
$$

Below we put shunting rule (20) at work in the derivation of a *trading*-rule which will enable handling composite antecedent and consequent functions in FDs:

$$g \xleftarrow{h \cdot R \cdot k^\circ} f \;\equiv\; g \cdot h \xleftarrow{R} f \cdot k \tag{21}$$

Thanks to (20), the calculation of (21) is immediate:

$$
\begin{array}{ll}
& g \xleftarrow{h \cdot R \cdot k^\circ} f \\
\equiv & \quad \{\text{definition (13); converses}\} \\
& g \leqslant f \cdot k \cdot R^\circ \cdot h^\circ \\
\equiv & \quad \{\text{shunting rule (20)}\} \\
& g \cdot h \leqslant (f \cdot k) \cdot R^\circ \\
\equiv & \quad \{\text{definition (13)}\} \\
& g \cdot h \xleftarrow{R} f \cdot k \qquad \square
\end{array}
$$

Another result about relational injectivity which will help in the sequel is

$$X \leqslant R \cup S \quad \equiv \quad X \leqslant R \;\wedge\; X \leqslant S \;\wedge\; R^\circ \cdot S \subseteq \ker X \tag{22}$$

where $R \cup S$ is the union of relations R and S. For $X := id$, (22) tells that $R \cup S$ is injective *iff* both R and S are injective and *don't "equivocate" each other*: wherever bSa and bRc hold, $c = a$. The calculation of (22) follows:

$$
\begin{array}{ll}
& X \leqslant R \cup S \\
\equiv & \quad \{\text{definitions of } \leqslant \text{ (12) and kernel (11)}\} \\
& (R \cup S)^\circ \cdot (R \cup S) \subseteq \ker X \\
\equiv & \quad \{\text{distribution of converse and composition over union}\} \\
& (R^\circ \cdot R) \cup (R^\circ \cdot S) \cup (S^\circ \cdot R) \cup (S^\circ \cdot S) \subseteq \ker X \\
\equiv & \quad \{\text{kernel (11)}\} \\
& \ker R \cup (R^\circ \cdot S) \cup (S^\circ \cdot R) \cup \ker S \subseteq \ker X \\
\equiv & \quad \{\text{universal property: } R \cup S \subseteq X \equiv R \subseteq X \;\wedge\; S \subseteq X;\ (12)\} \\
& X \leqslant R \;\wedge\; R^\circ \cdot S \subseteq \ker X \;\wedge\; S^\circ \cdot R \subseteq \ker X \;\wedge\; X \leqslant S \\
\equiv & \quad \{\text{the intermediate conjuncts are the same (taking converses)}\} \\
& X \leqslant R \;\wedge\; R^\circ \cdot S \subseteq \ker X \;\wedge\; X \leqslant S \qquad \square
\end{array}
$$

Hoare logic counterparts Galois connection (20) holds with no further change once $\leqslant$ is replaced by the preorder adopted for Hoare triples (16), the reasoning being the same. Fact (22) is even simpler for such a preorder, as the third conjunct

disappears.[11] Finally, the Hoare logic counterpart of (21) is

$$q \xleftarrow{h \cdot R \cdot k^\circ} p \quad \equiv \quad wp(h,q) \xleftarrow{R} wp(k,p) \tag{23}$$

where $wp(f,p) = \mathsf{dom}\,([\![p]\!] \cdot f)$ denotes the weakest-precondition for *function* f to ensure p on the output.[12] The first steps of the proof of (23) are the same as those of (21), leading to $q \cdot h \leqslant p \cdot k \cdot R^\circ$ (abbreviating $[\![p]\!]$, $[\![q]\!]$ to p, q). But the calculation requires further reasoning in this case because predicates p, q are (relationally) partial identities (tests) and therefore are not at the same level as functions. Since domain is idempotent, dom can be added to any of R or S in $R \leqslant S$ and thus dom is a self-adjoint concerning (16):

$$\mathsf{dom}\,R \leqslant S \quad \equiv \quad R \leqslant \mathsf{dom}\,S \tag{24}$$

Then:

$$\begin{array}{ll} & q \cdot h \leqslant \mathsf{dom}\,(p \cdot k \cdot R^\circ) \\ \equiv & \{\text{domain: } \mathsf{dom}\,(R \cdot S) = \mathsf{dom}\,(\mathsf{dom}\,R \cdot S)\} \\ & q \cdot h \leqslant \mathsf{dom}\,(\mathsf{dom}\,(p \cdot k) \cdot R^\circ) \\ \equiv & \{(24)\} \\ & \mathsf{dom}\,(q \cdot h) \leqslant \mathsf{dom}\,(p \cdot k) \cdot R^\circ \\ \equiv & \{\text{weakest precondition for functions: } wp(f,p) = \mathsf{dom}\,(p \cdot f)\} \\ & wp(h,q) \leqslant wp(k,p) \cdot R^\circ \\ \equiv & \{\text{Hoare triple (15)}\} \\ & wp(h,q) \xleftarrow{R} wp(k,p) \qquad \square \end{array}$$

5. Building a type system of FDs

The machinery set up in the previous sections is enough for developing a type system whereby *dependencies, relations and operations on relations are tied together*, as envisaged by Beeri et al. [9].

Composition rule FDs on relations which matching antecedent and consequent functions (as types) compose:

$$y \xleftarrow{S \cdot R} x \quad \Leftarrow \quad y \xleftarrow{S} z \quad \wedge \quad z \xleftarrow{R} x \tag{25}$$

Proof.

$$\begin{array}{ll} & h \xleftarrow{S} g \quad \wedge \quad g \xleftarrow{R} f \\ \equiv & \{(13)\ \text{twice}\} \\ & h \leqslant g \cdot S^\circ \quad \wedge \quad g \leqslant f \cdot R^\circ \\ \Rightarrow & \{\leqslant\text{-monotonicity of } (\cdot S^\circ);\ \text{converse (9)}\} \\ & h \leqslant g \cdot S^\circ \quad \wedge \quad g \cdot S^\circ \leqslant f \cdot (S \cdot R)^\circ \\ \Rightarrow & \{\leqslant\text{-transitivity}\} \\ & h \leqslant f \cdot (S \cdot R)^\circ \\ \equiv & \{(13)\ \text{again}\} \\ & h \xleftarrow{S \cdot R} f \qquad \square \end{array}$$

For R and S the same database table (tuple set), this rule subsumes Armstrong axiom F5 (Transitivity) in the standard FD theory [4]. For R and S regarded as describing computations, rule (25) is the FD counterpart of the rule of composition in Hoare logic, recall (17).[13]

[11] This happens because dom distributes through union, while ker does not. Both versions of (20) are instances of a generic result concerning *Galois connection lifting*, see appendix D.6 of [14].

[12] Terms such as $h \cdot R \cdot k^\circ$ denote programs which begin by reversing a function, proceeding as R and then updating the state by another function; for instance, program `{x := x-1; R; x := 2*x}` on a single-variable state `x` is denoted by relational term $(2*) \cdot R \cdot (1+)^\circ$, for some subprogram R.

[13] The proof is the same, as both (12) and (16) are preorders (thus transitive) compatible with relational composition. Recall that $R;S = S \cdot R$.

Consequence (weakening/strengthening) rule

$$k \xleftarrow{R} h \quad \Leftarrow \quad k \leqslant g \ \wedge \ g \xleftarrow{R} f \ \wedge \ f \leqslant h \tag{26}$$

Proof. See [14], where this rule is shown to subsume and generalize standard Armstrong axioms F2 (*Augmentation*) and F4 (*Projectivity*). In the parallel with Hoare logic, it corresponds to the two *rules of consequence* [11] which, put together and writing triples as arrows, becomes

$$q' \xleftarrow{R} p' \quad \Leftarrow \quad q' \Leftarrow q \ \wedge \ q \xleftarrow{R} p \ \wedge \ p \Leftarrow p'$$

for a program R and assertions p, q, p', q'. (Note that implications $q' \Leftarrow q$ etc. correspond to $[\![q]\!] \subseteq [\![q']\!]$ and therefore to $q' \leqslant q$ once brackets $[\![\,]\!]$ are dropped for simplicity.)

Reflexivity We have seen already that

$$f \xleftarrow{id} f \tag{27}$$

holds trivially. This rule, which corresponds to the "*skip*" rule of Hoare logic, $p \xleftarrow{skip} p$ is easily shown to hold for any set T,

$$f \xleftarrow{[\![T]\!]} f \tag{28}$$

as FDs are downward closed (that is, preserved by sub-relations, by monotonicity). Rule (28) is known as Armstrong axiom F1 (*Reflexivity*).

Note in passing that (25) and (27) together define a *category* whose objects are functions (types) and whose morphisms (arrows) are FDs.

6. Type checking database operations

Merging (union) Let us proceed to an example of database operation *type checking*: we want to know what it means for the merging of two database files to satisfy a particular functional dependency $f \longrightarrow g$. That is, we want to find a *sufficient* condition for the union $R \cup S$ of two relations R and S to be of type $f \longrightarrow g$. The relational algebra of injectivity does most of the work:

$$\begin{array}{ll} & g \xleftarrow{R \cup S} f \\ \equiv & \quad \{\text{definition (13); converse distributes by union}\} \\ & g \leqslant f \cdot (R^\circ \cup S^\circ) \\ \equiv & \quad \{\text{relational composition distributes through union}\} \\ & g \leqslant f \cdot R^\circ \cup f \cdot S^\circ \\ \equiv & \quad \{\text{algebra of injectivity (22); definition (13) again, twice}\} \\ & g \xleftarrow{R} f \ \wedge \ g \xleftarrow{S} f \ \wedge \ R \cdot \ker f \cdot S^\circ \subseteq \ker g \\ \equiv & \quad \{\text{introduce "mutual dependency" shorthand}\} \\ & g \xleftarrow{R} f \ \wedge \ g \xleftarrow{S} f \ \wedge \ g \xleftarrow{R,S} f \qquad \square \end{array}$$

The "mutual dependency" shorthand $g \xleftarrow{R,S} f$ introduced in the last step for $R \cdot \ker f \cdot S^\circ \subseteq \ker g$ can be read as a generalization of the standard definition of FD to *two* relations instead of *one* — just generalize the second R in (8) to some S. For R and S two sets of tuples, it means that grabbing one tuple from one set and another tuple from the other set, if they cannot be distinguished by f then they will remain indistinguishable by g.

It should be stressed that the bottom line of the calculation expresses not only a *sufficient* but also a *necessary* condition for $g \xleftarrow{R \cup S} f$ to hold, as all steps are equivalences. Summing up, rule

$$g \xleftarrow{R \cup S} f \quad \equiv \quad g \xleftarrow{R} f \ \wedge \ g \xleftarrow{S} f \ \wedge \ g \xleftarrow{R,S} f \tag{29}$$

holds.[14]

[14] The counterpart of (29) in Hoare logic is $\{p\}R\{q\} \ \wedge \ \{p\}S\{q\} \equiv \{p\}(R \cup S)\{q\}$ written directly in the original triple notation, where $R \cup S$ denotes the non-deterministic choice between R and S.

Type checking other database operations will follow the same scheme. Below we handle in detail one particular such operation, *relational join* [4]. This is justified not only for its relevance in data processing but also because it brings about other standard FD rules not yet addressed.

Joining (pairing) Recall from Section 1 how [3] explains the relevance of Tarski's work on *pairing* in relation algebra by illustrating how a ternary (in general, n-ary) relation $\{(a,b,c),(d,e,f)\}$ gets represented by a binary one, $\{(a,(b,c)),(d,(e,f))\}$.

Pairing is not only useful for ensuring that sets of arbitrarily long (but finite) tuples are representable by binary relations but also for defining the *join* operator ($\vartriangle$) on such sets. This operator turns out to be particularly handy to formalize in case the two sets of tuples are already represented as relations R and S:

$$(a,b)(R \vartriangle S)c \quad \equiv \quad a\ R\ c\ \wedge\ b\ S\ c \tag{30}$$

Interestingly, relational join behaves as a *least upper bound* with respect to the injectivity preorder:

$$R \vartriangle S \leqslant T \quad \equiv \quad R \leqslant T\ \wedge\ S \leqslant T \tag{31}$$

This arises from fact[15]

$$\ker(R \vartriangle S) \quad = \quad \ker R \cap \ker S \tag{32}$$

as follows:

$$\begin{array}{ll} & R \vartriangle S \leqslant T \\ \equiv & \quad \{(12) \text{ and } (32)\} \\ & \ker T \subseteq (\ker R) \cap (\ker S) \\ \equiv & \quad \{\text{universal property: } X \subseteq R \cap S \equiv X \subseteq R\ \wedge\ X \subseteq S\} \\ & \ker T \subseteq \ker R\ \ \wedge\ \ \ker T \subseteq \ker S \\ \equiv & \quad \{(12) \text{ twice}\} \\ & R \leqslant T\ \ \wedge\ \ S \leqslant T \qquad \square \end{array}$$

This combinator, termed *split* in [23], *fork* in [24] and *strict fork* in [1], turns out to be more general than its use in data processing suggests. In particular, when R and S are functions f and g, $f \vartriangle g$ is the obvious function which pairs the outputs of f and g: $(f \vartriangle g)x = (f(x), g(x))$. Think for instance of the projection function f_x (resp. f_y) which, in the context of Definition 1 yields $t[x]$ (resp. $t[y]$) when applied to a tuple t. Then $(f_x \vartriangle f_y)t = (t[x], t[y]) = t[xy]$, where xy denotes the union of attributes x and y [4]. So, attribute union corresponds to joining the corresponding projection functions. This gives us a quite uniform framework for handling both relational join and compound attributes. To make notation closer to what is common in data dependency theory we will abbreviate $f_x \vartriangle f_y$ to $f_x f_y$ and this even further to xy, identifying (as we did before) each attribute (say x) with the corresponding projection function (say f_x).

Keeping abbreviation fg of $f \vartriangle g$ (for functions), from (31) it is easy to derive facts $! \leqslant f \leqslant id$, $f \leqslant fg$ and $g \leqslant fg$. This is consistent with the use of juxtaposition to denote "sets of attributes". Likewise, $\leqslant$ can be regarded as expressing "attribute inclusion" in this context: the more attributes one observes the more injective the projection function corresponding to such attributes is.[16]

A first illustration of this unified framework is given below: the (generic) calculation of the so-called Armstrong axioms F3 (*Additivity*) and F4 (*Projectivity*).[17] This is done in one go, for arbitrary (suitably typed) R, f, g, h[18]:

$$gh \xleftarrow{R} f \quad \equiv \quad g \xleftarrow{R} f \quad \wedge \quad h \xleftarrow{R} f \tag{33}$$

Calculation:

$$\begin{array}{ll} & gh \xleftarrow{R} f \\ \equiv & \quad \{(13); \text{ expansion of shorthand } gh\} \\ & g \vartriangle h \leqslant f \cdot R^{\circ} \end{array}$$

[15] Fact (32) follows immediately from $(R \vartriangle S)^{\circ} \cdot (X \vartriangle Y) = R^{\circ} \cdot X \cap S^{\circ} \cdot Y$ [23].

[16] Note how ! mimics the empty set and id mimics the whole set of attributes, enabling one to "see the whole thing" and thus discriminating as much as possible.

[17] See [4].

[18] In the Hoare logic counterpart of this rule, gh stands for the product $g \times h$ of predicates g and h defined by $(g \times h)(b,a) = g(b)\ \wedge\ h(a)$. The rule, derived in [15], ensures that the category of FDs has products.

$$
\begin{array}{ll}
\equiv & \{\text{universal property of } \triangle \text{ (31)}\} \\
& g \leqslant f \cdot R^\circ \ \wedge \ h \leqslant f \cdot R^\circ \\
\equiv & \{\text{(13) twice}\} \\
& g \xleftarrow{R} f \ \wedge \ h \xleftarrow{R} f \qquad \square
\end{array}
$$

The typing rule for the join $R \triangle S$ of two relations R and S is calculated in the same way. The reasoning involves properties of the projection functions $\pi_1(x, y) = x$ and $\pi_2(x, y) = y$:

$$
\begin{array}{ll}
& g \xleftarrow{R} f \ \wedge \ h \xleftarrow{S} f \\
\Rightarrow & \{\pi_1 \cdot (R \triangle S) \subseteq R \text{ and } \pi_2 \cdot (R \triangle S) \subseteq S;\ \text{FDs are downward closed}\} \\
& g \xleftarrow{\pi_1 \cdot (R \triangle S)} f \ \wedge \ h \xleftarrow{\pi_2 \cdot (R \triangle S)} f \\
\equiv & \{\text{trading (21) twice}\} \\
& g \cdot \pi_1 \xleftarrow{R \triangle S} f \ \wedge \ h \cdot \pi_2 \xleftarrow{R \triangle S} f \\
\equiv & \{\text{F3} + \text{F4 (33)}\} \\
& (g \cdot \pi_1) \triangle (h \cdot \pi_2) \xleftarrow{R \triangle S} f \\
\equiv & \{\text{product of functions: } f \times g = (f \cdot \pi_1) \triangle (g \cdot \pi_2)\} \\
& g \times h \xleftarrow{R \triangle S} f \qquad \square
\end{array}
$$

7. Beyond the type system: database operation optimization

As explained above, FD theory (resp. Hoare logic) can be regarded as a type system whose rules help in reasoning about data models (resp. programs) without going into the semantic intricacies of data business rules (resp. program meanings).

When compared to the quantified expression of Definition 1, quantifier-free equivalent (13) looks simpler and is therefore expected to be easier to use in practice. This section gives two illustrations of this, one concerned with query optimization and the other with optimizing lexicographic sorting of database files.

FDs for query optimization This example, taken from [5], is also addressed by [25]: one wants to optimize the conjunctive query

$$\left\{(d, a') \mid (t, d, a) \in \mathit{Movies},\ (t', d', a') \in \mathit{Movies},\ t = t'\right\} \tag{34}$$

over a database file *Movies*(*Title*, *Director*, *Actor*) into a query accessing this file only once, knowing that FD *Title* $\rightarrow$ *Director* holds.

Using abbreviations M, t, d and a for (respectively) *Movies*, *Title*, *Director* and *Actor*, we want to solve for X the equation

$$d \cdot M \cdot (\ker t) \cdot M \cdot a^\circ = X \tag{35}$$

— whose left hand side is the relational equivalent of (34) [19] — aiming at a solution X containing only one instance of M. The equation is solved by taking FD $d \xleftarrow{M} t$ itself as starting point and trying to re-write it into something one recognizes as an instance of (35):

$$
\begin{array}{ll}
& d \xleftarrow{M} t \\
\equiv & \{(13)\} \\
& d \leqslant t \cdot M^\circ \\
\equiv & \{\text{expanding (11), (12); } M^\circ = M \text{ since } M \text{ is a partial identity}\} \\
& M \cdot t^\circ \cdot t \cdot M \subseteq d^\circ \cdot d \\
\equiv & \{\text{composition } (\cdot M) \text{ with a partial identity [14]}\} \\
& M \cdot t^\circ \cdot t \cdot M \subseteq d^\circ \cdot d \cdot M
\end{array}
$$

[19] As the interested reader may check by introducing the variables back. Note how $\ker t$ expresses $t = t'$ and projection functions d (for *Director*) and a (for *Actor*) work over tuple (t, d, a) and tuple (t', d', a'), respectively. The use of the same letters for data variables and the corresponding projection functions should help in comparing the two versions of the query.

$$\Rightarrow \quad \{\text{shunting (18), (19); monotonicity of } (\cdot a^\circ)\text{; kernel (11)}\}$$
$$d \cdot M \cdot (\ker t) \cdot M \cdot a^\circ \subseteq d \cdot M \cdot a^\circ \qquad \square$$

We thus find $d \cdot M \cdot a^\circ$ as a candidate solution for X. To obtain $X = d \cdot M \cdot a^\circ$ it remains to check the converse inclusion:

$$d \cdot M \cdot a^\circ \subseteq d \cdot M \cdot (\ker t) \cdot M \cdot a^\circ$$
$$\Leftarrow \quad \{id \subseteq \ker t \text{ because kernels of functions are equivalence relations}\}$$
$$d \cdot M \cdot a^\circ \subseteq d \cdot M \cdot M \cdot a^\circ$$
$$\equiv \quad \{M \cdot M = M \cap M = M \text{ because } M \text{ is a partial identity}\}$$
$$d \cdot M \cdot a^\circ \subseteq d \cdot M \cdot a^\circ \qquad \square$$

Altogether, FD $d \xleftarrow{M} t$ grants the solution $X = d \cdot M \cdot a^\circ$ to Eq. (35) — that is

$$X = \{(d, a') \mid (t, d, a') \in Movies\}$$

— which optimizes the given query by only visiting the movies file once.[20]

Optimizing lexicographic sorting This example calculates an improvement in lexicographic sorting of database files subject to FDs. Let $\leqslant$ and $\sqsubseteq$ be two preorders of the same type. By the expression $\leqslant \triangleright \sqsubseteq$ we mean the lexicographic order

$$a(\leqslant \triangleright \sqsubseteq)a' \quad \equiv \quad a \leqslant a' \ \wedge \ (a \geqslant a' \Rightarrow a \sqsubseteq a')$$

which gives priority to $\leqslant$, that is,

$$\leqslant \triangleright \sqsubseteq \quad = \quad \leqslant \cap (\leqslant^\circ \Rightarrow \sqsubseteq) \tag{36}$$

where relational implication is the upper adjoint of intersection:

$$R \cap S \subseteq X \quad \equiv \quad R \subseteq (S \Rightarrow X) \tag{37}$$

Now suppose that T is a database file whose schema includes attribute x (resp. y) whose domain is ordered by partial order $\leqslant_x$ (resp. $\leqslant_y$). Thus the tuples of T can be ordered not only by the preorders

$$t \leqslant_a^T t' \quad \equiv \quad t, t' \in T \ \wedge \ a(t) \leqslant_a a(t)' \tag{38}$$

for $a \in \{x, y\}$, but also by lexicographic combinations thereof, e.g. $\leqslant_x^T \triangleright \leqslant_y^T$. However, such lexicographic preorders can be simplified in presence of FDs. Below we calculate a sufficient condition for such a lexicographic preorder to reduce to one of its components, for instance:

$$\leqslant_x^T \triangleright \leqslant_y^T \quad = \quad \leqslant_x^T \quad \Leftarrow \quad y \xleftarrow{T} x \tag{39}$$

The relation-algebraic calculation of rule (39) goes in the same style as before[21]:

$$\leqslant_x^T \triangleright \leqslant_y^T \quad = \quad \leqslant_x^T$$
$$\equiv \quad \{(36);\ X \cap Y = X \text{ equivalent to } X \subseteq Y\}$$
$$\leqslant_x^T \subseteq ((\leqslant_x^T)^\circ \Rightarrow \leqslant_y^T)$$
$$\equiv \quad \{\text{Galois connection (37)}\}$$
$$\leqslant_x^T \cap (\leqslant_x^T)^\circ \subseteq \leqslant_y^T$$
$$\equiv \quad \{\text{variable-free versions of (38) for } a \in \{x, y\}\text{; converses}\}$$
$$T \cdot x^\circ \cdot \leqslant_x \cdot x \cdot T \cap T \cdot x^\circ \cdot \leqslant_x^\circ \cdot x \cdot T \subseteq T \cdot y^\circ \cdot \leqslant_y \cdot y \cdot T$$
$$\equiv \quad \{\text{distributions over intersection, as } x \cdot T \text{ is univalent; converses}\}$$
$$T \cdot x^\circ \cdot (\leqslant_x \cap \leqslant_x^\circ) \cdot x \cdot T \subseteq T \cdot y^\circ \cdot \leqslant_y \cdot y \cdot T$$

[20] By the way: symmetry between a and d in calculation step $d \cdot M \cdot t^\circ \cdot t \cdot M \cdot a^\circ \subseteq d \cdot M \cdot a^\circ$ above immediately tells that FD $a \xleftarrow{M} t$ would also enable the proposed optimization.

[21] The fourth step in the reasoning relies on prop. 5.3 of [1]: for univalent Q, distribution law $(R \cap S) \cdot Q = (R \cdot Q) \cap (S \cdot Q)$ holds — and therefore (taking converses) so does $Q^\circ \cdot (R \cap S) = (Q^\circ \cdot R) \cap (Q^\circ \cdot S)$.

$$
\begin{array}{ll}
\equiv & \{\leqslant_x \text{ is antisymmetric: } \leqslant_x \cap \leqslant_x^\circ = id\} \\
& T \cdot x^\circ \cdot x \cdot T \subseteq y^\circ \cdot \leqslant_y \cdot y \\
\Leftarrow & \{\leqslant_y \text{ is reflexive}\} \\
& T \cdot x^\circ \cdot x \cdot T \subseteq y^\circ \cdot y \\
\equiv & \{(6); (13)\} \\
& y \xleftarrow{T} x \qquad \square
\end{array}
$$

8. Conclusions

> "*The great merit of algebra is as a powerful tool for exploring family relationships over a wide range of different theories.* (...) *It is only their algebraic properties that emphasize the family likenesses* (...) *Algebraic proofs by term rewriting are the most promising way in which computers can assist in the process of reliable design.*"
>
> [Hoare and Jifeng [13]]

There is a growing interest in algebraic reasoning in computer science able to eventually promote calculational techniques in software engineering, hopefully unifying seemingly disparate theories once they are encoded into the same abstractions. Relation algebra [1] is particularly apt in this respect.

The current paper shows how a relation-algebraic rendering of both data dependency theory and Hoare logic purports one such unification, in spite of the latter being an algorithmic theory and the former a data theory, thanks to both algorithms and data structures being expressed in the *unified language of binary relations*.

In short (and informally), both logics rely on triples: *something* (a data set; a program) lies between an *antecedent* and a *consequent* observation (a data attribute; a state assertion); there is an ordering (injectivity; definition) on observations; triples express that antecedent observations are "enough" for consequents to hold "modally through" what is in between.

Triples are nicely captured by *arrows*, whose end-points can be regarded as *types*. On the data side, our approach equips relational data with *functional types* and an associated type system which can be used to *type check* database operations and optimize queries by calculation once they are written as Tarskian, quantifier-free formulas.

As formal verification is becoming more and more widespread to ensure quality in complex software systems, we believe our approach may contribute to *unified* formal verification tools blending in the same framework extended static checking and database programming.

Back to the opening story, surely Tarski's work on satisfaction and truth is relevant to computer science. But Etchemendy's answer could have been better tuned to the particular context of database technology suggested by the Oracle towers landscape:

> [...] "*They would never have been built without Tarski's work on the calculus of binary relations.*"

9. Related and future work

Functional dependencies have been characterized relationally by checking the determinism of the relations obtained by projecting tuple sets by antecedent and consequent attributes [26,27]. This alternative definition is equivalent to the one followed in the current paper.[22] As a generalization, Jaoua et al. [27] also study so-called *difunctional* dependencies.

Dependencies in relational databases have also been expressed using so-called *indiscernibility relations* [28]. Freyd and Scedrov [16] develop a τ-category theory of relations based on *monic n-tuples*. Concepts such as *table*, *column*, *short column* etc. fit into the spirit of (pointfree) data dependency and database theory and should be carefully studied in the context of the current paper.

Wisnesky [25] addresses the semantic optimization of monad comprehensions in functional programming by generalizing results from relational database theory. As this theory relies on the *powerset* monad, whose comprehensions correspond to database queries, by handling similar optimizations in relation algebra (as we did in Section 7) we have followed the well-known shift towards the Kleisli adjoint category.

As is well-known, this shift can be generalized to any other monad. Wisnesky [25] includes queries on probabilistic databases, this time relying on the (finite support) *distribution* monad. As shown by Oliveira [29], the "Kleisli shift" w.r.t. this monad leads to typed *linear algebra*. Wong and Butz [30] introduce Bayesian embedded multivalued dependencies as necessary and sufficient conditions for lossless decomposition of probabilistic relations. Lossless decomposition and multivalued dependencies have been handled by Oliveira [14] in the same way as FDs in the current paper. The prospect of calculating with data dependencies in probabilistic systems *directly* in matrix algebra is an interesting prospect for future

[22] See section *Generic relational projections* in [14].

work, in line with the consistent use of matrix notation by Schmidt [1] in relation algebra. Whether this carries over to probabilistic Hoare logic [31] remains to be seen.

Other ways of relating data dependency theory with algorithmic reasoning can be devised. For instance, Mili et al. [12] reason about while-loops $w = (\textbf{while}\ t\ \textbf{do}\ b)$ in terms of so-called *strongest invariant functions*, where invariant functions f, ordered by injectivity, are such that $f \cdot [\![t]\!] = f \cdot b \cdot [\![t]\!]$ holds. A simple argument in relation algebra shows this equivalent to $f \cdot b \cdot [\![t]\!] \subseteq f$, thus entailing FD $f \xleftarrow{b \cdot [\![t]\!]} f$. How much of our FD relation-algebraic approach could be applied in this setting is open to research. This includes finding a meaningful counterpart of the *rule of iteration* [11] at data level, a topic not addressed in the current paper.

Another law not considered in the correspondence between Hoare logic and functional dependencies is the *axiom of assignment*, $\{p[e/x]\}x := e\{p\}$ where $p[e/x]$ denotes the predicate which is obtained from p by replacing all occurrences of x by e. This axiom is interesting because it relies on the state structure of imperative programs: variables which hold data. Assignment $x := e$ means *selective updating*: program variable x is updated to e. A possible data-level counterpart to such selective updating is the SQL `update` command, which is of the form `UPDATE` R `SET` f `WHERE` x, meaning: *update all tuples in R which satisfy selection criterion x by tuple-transformation f, leaving the rest unchanged*. While the semantics of this operation is easy to encode in relation algebra, the parallel is somewhat artificial and needs further analysis. In general, future work should identify which generic properties of the $\leqslant$ relation on types are common to both frameworks and derive a more general kernel theory which both are instances of.

Last but not least, another prospect for future work concerns automated reasoning. RelView [32] is a well-known system that calculates with relations "beyond toy size". Many applications of relation algebra have been handled successfully in this tool. Algebraic structures such as idempotent semirings and Kleene algebras (which relation algebra is an instance of) have also been shown to be amenable to automation by e.g. Höfner and Struth [8] and Struth [7]. Möller et al. [33] encode a database preference theory into idempotent semiring algebra and show how to use Prover9 to discharge proofs. Model checking of extended static checks in tools such as e.g. the Alloy Analyser also blends well with algebraic-relational models [34].

The implementation of a generic, unified approach to both data and program theories on top of libraries already available in such automated deduction systems is a prospect for long term research.

Acknowledgements

This work is funded by ERDF – European Regional Development Fund through the COMPETE Programme (operational programme for competitiveness) and by National Funds through the *FCT – Fundação para a Ciência e a Tecnologia* (Portuguese Foundation for Science and Technology) within project FCOMP-01-0124-FEDER-028923.

The author would like to thank the anonymous referees for insightful comments which significantly improved the quality of the original submission. Feedback and exchange of ideas with Ryan Wisnesky about pointfree query reasoning are also gratefully acknowledged.

References

[1] G. Schmidt, Relational Mathematics, in: Encyclopedia of Mathematics and Its Applications, vol. 132, Cambridge University Press, 2010.
[2] S. Feferman, Tarski's influence on computer science, Log. Methods Comput. Sci. 2 (2006), 1–1–13.
[3] J. Bussche, Applications of Alfred Tarski's ideas in database theory, in: CSL'01: Proceedings of the 15th International Workshop on Computer Science Logic, Springer-Verlag, London, UK, 2001, pp. 20–37.
[4] D. Maier, The Theory of Relational Databases, Computer Science Press, 1983.
[5] S. Abiteboul, R. Hull, V. Vianu, Foundations of Databases, Addison–Wesley, 1995.
[6] A. Tarski, S. Givant, A Formalization of Set Theory without Variables, AMS Colloquium Publications, vol. 41, American Mathematical Society, Providence, Rhode Island, 1987.
[7] G. Struth, Isabelle repository for relational and algebraic methods, URL: http://staffwww.dcs.shef.ac.uk/people/G.Struth/isa, 2011.
[8] P. Höfner, G. Struth, Automated reasoning in Kleene algebra, in: Proceedings, CADE-21, Springer-Verlag, 2007, pp. 279–294.
[9] C. Beeri, R. Fagin, J. Howard, A complete axiomatization for functional and multivalued dependencies in database relations, in: D. Smith (Ed.), Proc. 1977 ACM SIGMOD, Toronto, ACM, NY, USA, 1977, pp. 47–61.
[10] C. Flanagan, K. Leino, M. Lillibridge, G. Nelson, J. Saxe, R. Stata, Extended static checking for Java, in: PLDI, 2002, pp. 234–245.
[11] C. Hoare, An axiomatic basis for computer programming, Commun. ACM 12 (10) (1969) 576–580, 583.
[12] A. Mili, J. Desharnais, J. Gagné, Strongest invariant functions: Their use in the systematic analysis of while statements, Acta Inform. 22 (1985) 47–66.
[13] C. Hoare, H. Jifeng, Unifying Theories of Programming, Series in Computer Science, Prentice-Hall International, 1998.
[14] J. Oliveira, Pointfree foundations for (generic) lossless decomposition, Technical Report TR-HASLab:3:2011, HASLab/INESC TEC & U. Minho, 2011, URL: https://repositorium.sdum.uminho.pt/handle/1822/24648.
[15] J. Oliveira, Extended Static Checking by Calculation using the Pointfree Transform, LNCS, vol. 5520, Springer-Verlag, 2009, pp. 195–251.
[16] P. Freyd, A. Scedrov, Categories, Allegories, Mathematical Library, vol. 39, North-Holland, 1990.
[17] H. Doornbos, R. Backhouse, J. van der Woude, A calculational approach to mathematical induction, Theor. Comput. Sci. 179 (1997) 103–135.
[18] D. Kozen, Kleene algebra with tests, ACM Trans. Program. Lang. Syst. 19 (1997) 427–443.
[19] D. Kozen, On Hoare logic and Kleene algebra with tests, ACM Trans. Comput. Log. 1 (2000) 60–76.
[20] J. Desharnais, B. Möller, G. Struth, Kleene algebra with domain, ACM Trans. Comput. Log. 7 (2006) 798–833.
[21] I. Wehrman, C. Hoare, P.W. O'Hearn, Graphical models of separation logic, Inf. Process. Lett. 109 (2009) 1001–1004.
[22] C. Hoare, J. He, The weakest prespecification, Inf. Process. Lett. 24 (1987) 127–132.
[23] R. Bird, O. de Moor, Algebra of Programming, Series in Computer Science, Prentice-Hall International, 1997.
[24] M. Frias, G. Baum, A. Haeberer, Fork algebras in algebra, logic and computer science, Fundam. Inform. (1997) 1–25.

[25] R. Wisnesky, Minimizing Monad comprehensions, Technical Report TR-02-11, Harvard University, Cambridge, Massachusetts, 2011.
[26] G. Schmidt, T. Ströhlein, Relations and Graphs: Discrete Mathematics for Computer Scientists, EATCS Monographs on Theoretical Computer Science, Springer-Verlag, 1993.
[27] A. Jaoua, N. Belkhiter, H. Ounalli, T. Moukam, Databases, in: C. Brink, W. Kahl, G. Schmidt (Eds.), Relational Methods in Computer Science, Springer-Verlag New York, Inc., New York, NY, USA, 1997, pp. 197–210.
[28] H. Okuma, W. MacCaull, Y. Kawahara, Informational representability for contexts in Dedekind categories, Technical Report trcs208, Kyushu University, 2003, URL: http://www.i.kyushu-u.ac.jp/doitr/trcs208.ps.gz.
[29] J. Oliveira, Towards a linear algebra of programming, Form. Asp. Comput. 24 (2012) 433–458.
[30] S. Wong, C. Butz, The implication of probabilistic conditional independence and embedded multivalued dependency, in: 8th Conf. on Inf. Processing and Management of Uncertainty in K.-B. Systems, IPMU00, 2000, pp. 876–881.
[31] G. Barthe, B. Grégoire, S. Béguelin, Probabilistic relational Hoare logics for computer-aided security proofs, in: MPC'12, 2012, pp. 1–6.
[32] R. Berghammer, Computing and visualizing Banks sets of dominance relations using relation algebra and RelView, J. Log. Algebr. Program. 82 (2013) 123–136.
[33] B. Möller, P. Roocks, M. Endres, An algebraic calculus of database preferences, in: MPC 2012, in: LNCS, vol. 7342, Springer, Berlin, Heidelberg, 2012, pp. 241–262.
[34] J. Oliveira, M. Ferreira, Alloy meets the algebra of programming: A case study, IEEE Trans. Softw. Eng. 39 (2013) 305–326.