Functions as types or the "Hoare logic" of functional dependencies

José N. Oliveira

High Assurance Software Lab
INESC TEC and University of Minho
`jno@di.uminho.pt`

Abstract. Inspired by the trend on unifying theories of programming, this paper shows how the algebraic treatment of standard data dependency theory equips relational data with *functional types* and an associated type system which is useful for type checking database operations and for query optimization.
Such a *typed* approach to database programming is then shown to be of the same family as other programming logics such as eg. *Hoare logic* or that of *strongest invariant functions* which has been used in the analysis of while statements.
The prospect of using automated deduction systems such as Prover9 for type-checking and query optimization on top of such an algebraic approach is considered.

Keywords: Unifying theories of programming; theoretical foundations; data dependencies.

1 Prelude

In a paper addressing the influence of Alfred Tarski (1901-83) in computer science, Solomon Feferman [5] 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 [5]: *"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 [4] had in fact devoted attention to relating Codd and Tarski's work: *"We conclude that Tarksi 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 [4]:

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

To the best of the author's knowledge, nobody has thus far addressed this *exercise* in a thorough way. Instead, standard relational database theory [11, 1] 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. An effort which ultimately lead to his *formalization of set theory without variables* [17].

The topic has acquired recent interest with the advent of work on implementing extensions of Tarski's algebra in automated deduction systems such as *Prover9* and the associated counterexample generator *Mace4* [10]. 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 [2]:

"[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* [6] which are more and more regarded essential to ensuring quality in complex software systems.

In the remainder of this paper we will see how the algebraic treatment of the standard 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 eg. *Hoare logic* [8] 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 an *unifying theories of programming* [9] flavour, even though the exercise is not carried out "avant la lettre" in canonical UTP. A full account can be found in a technical report [15]. For space constraints, this paper only covers the first part of the exercise, that of developing a type system for relational data which stems from functional dependencies.

Paper structure. 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 is given in section 3. Section 4 shows that, in essence, *injectivity* is what matters in FDs and gives a corresponding, 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 section gives an account of related work and concludes with a prospect for future work.

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 file (vulg. *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, $T \subseteq A_1 \times \ldots \times A_n$. A *relational database* is just a collection of several such n-ary 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 $S = \{\textsc{Pilot}, \textsc{Flight}, \textsc{Date}, \textsc{Departs}\}$ concerning the airline scheduling system given as example in [11], 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 capture 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. [11] provides the following definition for FD-satisfiability:

Definition 1. *Given subsets* $x, y \subseteq S$ *of the relation scheme* S *of a* 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,*

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

(The notation $t[a]$ *in (1) means "the value exhibited by attribute* a *in tuple* t*".)*
□

How does one express formula (1) in Tarski's relation algebra style, getting way with 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:

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

Next, we observe that, given a function $f : A \rightarrow B$, 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 $a'(f^\circ \cdot f)a$.

[1] This is known as the *nucleous* [12] or *kernel* [14] of a function f.

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:

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

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 expressed as relation inclusions, following definition:

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

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),

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

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 [2]:

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

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

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

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).

[2] This is a standard way of encoding a set T as a binary relation $[\![T]\!]$ known as a *partial identity*, 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 $[\![S]\!] \cdot [\![T]\!] = [\![S]\!] \cap [\![T]\!]$.

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 nondeterministic 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:

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

In words: *inputs a, a' indistinguishable by f via R 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 the $f \to g$ FD, 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* [3]:

$$(R \cdot S)^\circ = S^\circ \cdot R^\circ \tag{9}$$
$$(R^\circ)^\circ = R \tag{10}$$

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

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

[3] 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.

and read $\mathsf{ker}\ R$ as "the *kernel* of R". Clearly, $a'(\mathsf{ker}\ R)a$ means $\exists\, b : b\ R\ a' \wedge b\ R\ a$ and therefore $\mathsf{ker}\ R$ 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*:

$$R \leq S \quad \triangleq \quad \mathsf{ker}\ S \subseteq \mathsf{ker}\ R \tag{12}$$

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 first loses more information (order and multiplicity) than the latter, which only forgets about order. Thus $elems \leq bagify$. A function f (relation in general) will be *injective* iff $\mathsf{ker}\ f \subseteq id$ ($id \leq f$), 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 [4], $R \leq S$ means that R *is less injective than* 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, $! \leq R \leq id$, where the infimum is captured by the constant function ! which maps every argument to a given (predefined) value, the choice of such value being irrelevant [5]. The kernel of ! is therefore the largest possible, denoted by $\top$ (for "top"). The other bound is trivial to check, since $\mathsf{ker}\ id = id$, this arising from the well-known fact that id is the unit of composition. In general, $id \leq R$ means 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):

$$\begin{array}{ll}
& f \xrightarrow{R} g \\
\equiv & \quad \{\ \text{definition (7)}\ \} \\
& R \cdot f^\circ \cdot f \cdot R^\circ \subseteq g^\circ \cdot g \\
\equiv & \quad \{\ \text{converses (9,10) ; kernel (11)}\ \} \\
& \mathsf{ker}\ (f \cdot R^\circ) \subseteq \mathsf{ker}\ g \\
\equiv & \quad \{\ \text{(12): } g \text{ is "\textit{less injective} than } f \text{ wrt. } R\text{"}\ \} \\
& g \leq f \cdot R^\circ
\end{array}$$

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

Definition 2. *Given an arbitrary binary relation $R \subseteq A \times B$ and functions $f : B \to D$ and $g : A \to C$, given A, B, ... D, the "type assertion"* $g \xleftarrow{R} f$

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

[5] Note that $R \leq S$ is a preorder, not a partial order, meaning that two relations indistinguishable with respect to their degree of injectivity can be different.

meaning that R satisfies FD $f \to g$ *is given by the equivalence:*

$$g \xleftarrow{R} f \quad \equiv \quad g \leq f \cdot R^\circ \tag{13}$$

□

Intuitively, $g \xleftarrow{R} f$ means that g will be *blinder* (less injective) to the outputs of R than f is concerning its inputs.

There are two main advantages in definition (13), besides saving ink. The most important is that it takes advantage of the 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"* [8].

Let program R be identified with the relation which captures its state transition semantics and predicates p (and q) be identified with $s' [\![p]\!] s \;\equiv\; s' = s \wedge p(s)$ (similarly for q) — the same trick we used for converting sets to binary relations in 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 this is captured by [6]

$$\{p\}R\{q\} \quad \equiv \quad rng(R \cdot [\![p]\!]) \subseteq [\![q]\!]$$

meaning that the outputs of R (given by the range operator rng) for inputs pre-conditioned by p don't fall outside q; that is, q is *weaker* than the strongest post-condition $sp(R, p)$, something we can express by writing

$$\{p\}R\{q\} \quad \equiv \quad q \leq p \cdot R^\circ \tag{14}$$

under a suitable preorder $\leq$ expressing that q is less constrained than $p \cdot R^\circ$ [7].

In spite of the different semantic context, there is a striking formal similarity between formulas (14) 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, where we are going to write $p \xrightarrow{R} q$ for $\{p\}R\{q\}$, to put the notations closer. Using this notation, rules such as eg. the rule of composition, $\{p\}R_1\{q\} \wedge \{q\}R_2\{r\} \Rightarrow \{p\}R_1; R_2\{r\}$ become [8]

$$p \xrightarrow{R_1} q \;\;\wedge\;\; q \xrightarrow{R_2} r \;\Rightarrow\; p \xrightarrow{R_1;R_2} r \tag{15}$$

[6] See [14] and references there to related work.

[7] Details: $\{p\}R\{q\}$ is $rng(R \cdot [\![p]\!]) \subseteq [\![q]\!]$, itself the same as $dom([\![p]\!] \cdot R^\circ) \subseteq dom [\![q]\!]$ since *dom* (domain) and *rng* (range) commute with converse and the domain of a partial identity is itself. The preorder is $R \leq S \equiv dom\ S \subseteq dom\ R$. Parentheses $[\![\_]\!]$ are dropped to make the formula lighter to read.

[8] The arrow notation for Hoare triples, reminiscent of that of labelled transition systems, is adopted in eg. [14].

We will check the FD equivalent to (15) shortly.

4 A calculus of injectivity ($\leq$)

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 [15]; the reader is referred to this technical report for the whole story.

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

$$f \cdot R \subseteq S \equiv R \subseteq f^\circ \cdot S \tag{16}$$

$$R \cdot f^\circ \subseteq S \equiv R \subseteq S \cdot f \tag{17}$$

In these equivalences [9], which are widely known as *shunting rules* [3], 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). It turns out that such rules are quite easy to infer, as is the case of the Galois connection for trading a function f with respect to the injectivity preorder given by

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

which takes just three steps to calculate:

$$\begin{array}{ll}
& R \cdot f \leq S \\
\equiv & \quad \{ \text{ definition (12) ; converses (9,10) ; kernel (11) } \} \\
& \mathsf{ker}\ S \subseteq f^\circ \cdot (\mathsf{ker}\ R) \cdot f \\
\equiv & \quad \{ \text{ shunting rules (16,17) } \} \\
& f \cdot \mathsf{ker}\ S \cdot f^\circ \subseteq \mathsf{ker}\ R \\
\equiv & \quad \{ \text{ converses, kernel and definition (12) again } \} \\
& R \leq S \cdot f^\circ
\end{array}$$

Let us put this new rule to work for us in the derivation of a *trading*-rule which will enable handling composite antecedent and consequent functions in FDs:

$$y \xleftarrow{z \cdot R \cdot k^\circ} x \ \equiv \ y \cdot z \xleftarrow{R} x \cdot k \tag{19}$$

[9] Technically, these equivalences should be regarded as (families of) Galois connections [14].

Thanks to (18), the calculation of (19) is immediate:

$$\begin{array}{ll}
 & y \xleftarrow{z\cdot R\cdot k^\circ} x \\
\equiv & \{ \text{ definition (13) ; converses } \} \\
 & y \leq x \cdot k \cdot R^\circ \cdot z^\circ \\
\equiv & \{ \text{ new shunting rule (18) } \} \\
 & y \cdot z \leq (x \cdot k) \cdot R^\circ \\
\equiv & \{ \text{ definition (13) } \} \\
 & y \cdot z \xleftarrow{R} x \cdot k
\end{array}$$

Another result which will help in the sequel is

$$X \leq R \cup S \;\equiv\; X \leq R \;\wedge\; X \leq S \;\wedge\; R^\circ \cdot S \subseteq \mathsf{ker}\, X \qquad (20)$$

where $R \cup S$ is the union of relations R and S [10]. For $X := id$, (20) tells that $R \cup S$ is injective **iff** both R and S are injective and *don't "confuse" each other*: wherever bSa and bRc hold, $c = a$.

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 [2] envisaged.

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

$$y \xleftarrow{S\cdot R} x \;\Leftarrow\; y \xleftarrow{S} z \;\wedge\; z \xleftarrow{R} x \qquad (21)$$

Proof:

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

[10] See [15] for the proof of this and other results of the algebra of injectivity.

$$\equiv \qquad \{ \ (13) \text{ again } \}$$

$$h \xleftarrow{S \cdot R} f$$

This rule, which for R and S the same database table subsumes Armstrong axiom F5 (Transitivity) in the standard FD theory [11], is the FD counterpart of the rule of composition in Hoare logic (15) for R and S regarded as describing computations.

Consequence (weakening/strengthening) rule:

$$k \xleftarrow{R} h \ \Leftarrow k \leq g \ \wedge \ \ g \xleftarrow{R} f \ \ \wedge \ f \leq h \tag{22}$$

Proof: See [15], 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* [8] which, put together and writing triples as arrows, becomes

$$q' \xleftarrow{P} p' \ \Leftarrow q' \Leftarrow q \ \wedge \ \ q \xleftarrow{P} p \ \ \wedge \ p \Leftarrow p'$$

for P a program and p, q etc program assertions.

Reflexivity. We have seen already that

$$f \xleftarrow{id} f \tag{23}$$

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,

$$f \xleftarrow{[\![T]\!]} f \tag{24}$$

as FDs are downward closed. Rule (24) is known as Armstrong axiom F1 (Reflexivity).

Note in passing that (21) and (23) together define a category whose objects are functions (types) and whose morphisms (arrows) are FDs.

6 Type checking database operations

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

[11] See [11]. The calculation of this and other similar results stated in this paper can be found in [15].

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 algebra of injectivity does most of the work:

$$
\begin{array}{ll}
& g \xleftarrow{R \cup S} f \\
\equiv & \{ \text{ definition (13) ; converse distributes by union } \} \\
& g \leq f \cdot (R^\circ \cup S^\circ) \\
\equiv & \{ \text{ relational composition distributes through union } \} \\
& g \leq f \cdot R^\circ \cup f \cdot S^\circ \\
\equiv & \{ \text{ algebra of injectivity (20); definition (13) again, twice } \} \\
& g \xleftarrow{R} f \quad \wedge \quad g \xleftarrow{S} f \quad \wedge \quad R \cdot \mathsf{ker}\ f \cdot S^\circ \subseteq \mathsf{ker}\ g \\
\equiv & \{ \text{ introduce “mutual dependency” shorthand } \} \\
& g \xleftarrow{R} f \quad \wedge \quad g \xleftarrow{S} f \quad \wedge \quad g \xleftarrow{R,S} f
\end{array}
$$

The “mutual dependency” shorthand $g \xleftarrow{R,S} f$ introduced in the last step for $R \cdot \mathsf{ker}\ f \cdot S^\circ \subseteq \mathsf{ker}\ g$ can be read as a generalization of the standard definition of a 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.

Type checking other database operations will follow the same scheme. Below we handle one particular such operation — relational *join* [11] — in detail. 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 and pairing. Recall from section 1 how [4] 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 represented by binary relations but also for defining the *join* operator ($\bowtie$) on such sets. In fact, this operator is particularly handy to express in case the two sets of tuples are already represented as binary relations R and S:

$$(a, b)(R \bowtie S)c \equiv a\ R\ c \wedge b\ S\ c \tag{25}$$

Interestingly, relational join behaves as a *least upper bound* with respect to the injectivity preorder [12]:

$$R \bowtie S \leq T \;\equiv\; R \leq T \;\wedge\; S \leq T \tag{26}$$

This combinator turns out to be more general than its use in data processing [13]. In particular, when R and S are functions f and g, $f \bowtie g$ is the obvious function which pairs the outputs of f and g: $(f \bowtie g)x = (f\ x, g\ x)$. 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 $(f_x \bowtie f_y)t = (t[x], t[y]) = t[xy]$, where xy denotes the union of attributes x and y [11]. 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 $f_x \bowtie f_y$ to $f_x f_y$ and this even further to xy, identifying (as we did before) each attribute (eg. x) with the corresponding projection function (eg. f_x).

Minding this abbreviation fg of $f \bowtie g$, for functions, from (26) it is easy to derive facts $! \leq f \leq id$ and $f \leq fg$, $g \leq fg$. This is consistent with the use of juxtaposition to denote "sets of attributes". In this context, $\leq$ can be regarded as expressing "attribute inclusion". In fact, the more attributes one observes the more injective the projection function corresponding to such attributes is [14].

A first illustration of this unified framework is given below: the (generic) calculation of the so-called Armstrong axioms F3 *(Additivity)* and F4 *(Projectivity)* [15]. This is done in one go, for arbitrary (suitably typed) R, f, g, h [16]:

$$gh \xleftarrow{R} f \;\equiv\; g \xleftarrow{R} f \;\;\wedge\;\; h \xleftarrow{R} f \tag{27}$$

Calculation:

$$\begin{array}{ll}
& gh \xleftarrow{R} f \\
\equiv & \quad \{\ (13)\ ;\ \text{expansion of shorthand } gh\ \} \\
& g \bowtie h \leq f \cdot R^{\circ} \\
\equiv & \quad \{\ \text{universal property of} \bowtie (26)\ \} \\
& g \leq f \cdot R^{\circ} \;\;\wedge\;\; h \leq f \cdot R^{\circ} \\
\equiv & \quad \{\ (13)\ \text{twice}\ \}
\end{array}$$

[12] Proof in [15].

[13] It is termed *split* in [3] and *fork* in [7].

[14] This parallel between attribute sets ordered by inclusion and projection functions ordered by injectivity is dealt with in detail in [15]. 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.

[15] See [11].

[16] In the Hoare logic counterpart of this rule gh will be the conjunction of two assertions.

$$g \xleftarrow{R} f \quad \wedge \quad h \xleftarrow{R} f$$

The type rule for the database join operator ($\bowtie$) is calculated in the same way:

$$g \xleftarrow{R} f \quad \wedge \quad h \xleftarrow{S} f$$

$\Rightarrow$ $\{$ let $\pi_1(y,x) = y$ and $\pi_2(y,x) = x$; FDs are downward closed $\}$

$$g \xleftarrow{\pi_1 \cdot (R \bowtie S)} f \quad \wedge \quad h \xleftarrow{\pi_2 \cdot (R \bowtie S)} f$$

$\equiv$ $\{$ trading (19) twice $\}$

$$g \cdot \pi_1 \xleftarrow{R \bowtie S} f \quad \wedge \quad h \cdot \pi_2 \xleftarrow{R \bowtie S} f$$

$\equiv$ $\{$ F3+F4 (27) $\}$

$$(g \cdot \pi_1) \bowtie (h \cdot \pi_2) \xleftarrow{R \bowtie S} f$$

$\equiv$ $\{$ product of functions: $f \times g = (f \cdot \pi_1) \bowtie (g \cdot \pi_2)$ $\}$

$$g \times h \xleftarrow{R \bowtie S} f$$

7 Beyond the type system: query optimization

As explained above, FD theory (cf. Hoare logic) can be regarded as a type system whose rules help in reasoning about data models (cf. programs) without going into the semantic intricacies of data business rules (cf. program meanings). It helps because quantified definitions such as definition 1 don't scale up very well to large sets of dependencies. In this respect, our quantifier-free equivalent (13) looks more tractable and is therefore expected to be calculationally effective where the quantified equivalent is clumsy.

This will be illustrated below with a simple example, taken from [1] and also addressed by [18]: one wants to optimize the conjunctive query

$$\{(d, a') \mid t = t', (t, d, a) \in \mathit{Movies}, (t', d', a') \in \mathit{Movies}\} \tag{28}$$

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

Put in calculational format and abbreviating M for Movies, t for Title, d for $\mathit{Director}$ and a for Actor, we want to solve for X the equation

$$d \cdot M \cdot (\mathsf{ker}\ t) \cdot M \cdot a^\circ = X \tag{29}$$

whose left hand side is the relational equivalent of (28) [17]. Our aim is to obtain a solution X containing only one instance of M. The equation is solved by

[17] As the interested reader may check by introducing the variables back. Note how $\mathsf{ker}\ t$ expresses $t = t'$ and projection functions d (for $\mathit{Director}$) and a (for Actor) work

taking the FD itself as starting point and trying to re-write it into something one recognizes as an instance of (29):

$$
\begin{array}{ll}
 & d \xleftarrow{M} t \\
\equiv & \quad \{ \ (13) \ \} \\
 & d \leq t \cdot M^\circ \\
\equiv & \quad \{ \ \text{expanding (11,12); } M^\circ = M \text{ since } M \text{ is a set} \ \} \\
 & M \cdot t^\circ \cdot t \cdot M \subseteq d^\circ \cdot d \\
\equiv & \quad \{ \ \text{composition } (\cdot M) \text{ with a set (partial identity) is a closure operator} \ \} \\
 & M \cdot t^\circ \cdot t \cdot M \subseteq d^\circ \cdot d \cdot M \\
\Rightarrow & \quad \{ \ \text{shunting (16,17); monotonicity of } (\cdot a^\circ)\text{; kernel (11)} \ \} \\
 & d \cdot M \cdot (\mathsf{ker}\ t) \cdot M \cdot a^\circ \subseteq d \cdot M \cdot a^\circ
\end{array}
$$

Thus we 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:

$$
\begin{array}{ll}
 & d \cdot M \cdot a^\circ \subseteq d \cdot M \cdot (\mathsf{ker}\ t) \cdot M \cdot a^\circ \\
\Leftarrow & \quad \{ \ id \subseteq \mathsf{ker}\ t \text{ because kernels are equivalence relations} \ \} \\
 & d \cdot M \cdot a^\circ \subseteq d \cdot M \cdot M \cdot a^\circ \\
\equiv & \quad \{ \ M \cdot M = M \cap M = M \text{ because } M \text{ is a set} \ \} \\
 & d \cdot M \cdot a^\circ \subseteq d \cdot M \cdot a^\circ
\end{array}
$$

Thus $X = d \cdot M \cdot a^\circ$, that is

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

is a solution to equation (29) which optimizes the given query by only visiting the movies file once [18].

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 tallying the two versions of the query.

[18] 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.

8 Conclusions and future work

"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 emphasise the family likenesses (...) Algebraic proofs by term rewriting are the most promising way in which computers can assist in the process of reliable design."

[9]

There is growing interest in applying abstract algebra techniques in computer science as a way to promote calculation in software engineering. Moreover, algebraic structures such as idempotent semirings and Kleene algebras (which relation algebra is an instance of) have been shown to be amenable to automation [10]. [13], for instance, encode a database preference theory into idempotent semiring algebra and show how to use Prover9 to discharge proofs. Model checking in tools such as eg. the Alloy Analyser also blends well with quantifier-free relational models [16].

Abstract algebra has the power to *unify* seemingly disparate theories once they are encoded into the same abstract terms. In the current paper we have shown how a relational rendering of both Hoare logic and data dependency theory purports one such unification, in spite of the former being an algorithmic theory and the latter a data theory, as both algorithms and data structures unify into binary relations.

Other such unifications could be devised. For instance, [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 $f \xleftarrow{b \cdot [\![t]\!]} f$.

On a more practical register, our algebraic framework makes it possible to type-check database operations and optimize queries by calculation once they are written as Tarskian, quantifier-free formulas. We would like to investigate this further in connection to [18]'s point-free query compiler.

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."*

Acknowledgments. This research was carried out in the context of the Mondrian Project funded by the Portuguese Science and Technology Foundation (FCT) contract PTDC/EIA-CCO/108302/2008. Feedback and exchange of ideas with Ryan Wisnesky about pointfree query reasoning are gratefully acknowledged.

References

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases.* Addison-Wesley, 1995. ISBN 0-201-53771-0.

[2] C. Beeri, R. Fagin, and J.H. Howard. A complete axiomatization for functional and multivalued dependencies in database relations. In D.C.P. Smith, editor, *Proc. 1977 ACM SIGMOD, Toronto*, pages 47–61, New York, NY, USA, 1977. ACM.

[3] R. Bird and O. de Moor. *Algebra of Programming.* Series in Computer Science. Prentice-Hall International, 1997.

[4] J. Bussche. Applications of Alfred Tarski's ideas in database theory. In *CSL '01: Proceedings of the 15th International Workshop on Computer Science Logic*, pages 20–37, London, UK, 2001. Springer-Verlag. ISBN 3-540-42554-3.

[5] S. Feferman. Tarski's influence on computer science. *Logical Methods in Computer Science*, 2:1–1–13, 2006. doi: 10.2168/LMCS-2(3:6)2006.

[6] C. Flanagan, K.R.M. Leino, M. Lillibridge, G. Nelson, J.B. Saxe, and R. Stata. Extended static checking for Java. In *PLDI*, pages 234–245, 2002.

[7] M.F. Frias, G. Baum, and A.M. Haeberer. Fork algebras in algebra, logic and computer science. *Fundam. Inform.*, pages 1–25, 1997.

[8] C.A.R. Hoare. An axiomatic basis for computer programming. *CACM*, 12,10:576–580, 583, October 1969.

[9] C.A.R. Hoare and He Jifeng. *Unifying Theories of Programming.* Series in Computer Science. Prentice-Hall International, 1998. C.A.R. Hoare, series editor.

[10] P. Höfner and G. Struth. Automated reasoning in Kleene algebra. In *Proceedings*, CADE-21, pages 279–294. Springer-Verlag, 2007. ISBN 978-3-540-73594-6.

[11] D. Maier. *The Theory of Relational Databases.* Computer Science Press, 1983. ISBN 0-914894-42-0.

[12] A. Mili, J. Desharnais, and J.R. Gagné. Strongest invariant functions: Their use in the systematic analysis of while statements. *Acta Inf.*, 22(1):47–66, 1985.

[13] B. Möller, P. Roocks, and M. Endres. An algebraic calculus of database preferences, 2012. Univ. of Augsburg, Germany (submitted).

[14] J.N. Oliveira. *Extended Static Checking by Calculation using the Pointfree Transform* . In A. Bove et al., editor, *LerNet ALFA Summer School 2008*, volume 5520 of *LNCS*, pages 195–251. Springer-Verlag, 2009.

[15] J.N. Oliveira. Pointfree foundations for (generic) lossless decomposition. Technical Report TR-HASLab:3:2011, HASLab, University of Minho and INESC TEC, 2011. available from `http://wiki.di.uminho.pt/twiki/bin/view/DI/FMHAS/TechnicalReports`.

[16] J.N. Oliveira and M.A. Ferreira. Alloy meets the algebra of programming: a case study, 2012. To appear in IEEE Transactions on Software Engineering.

[17] A. Tarski and S. Givant. *A Formalization of Set Theory without Variables*. American Mathematical Society, 1987. ISBN 0821810413. AMS Colloquium Publications, volume 41, Providence, Rhode Island.

[18] R. Wisnesky. A categorical query language, 2012. PhD thesis (due May 2012). Harvard University, School of Engineering and Applied Sciences.