A Crash Course on Database Queries

Non Peer-reviewed author version

VAN DEN BUSSCHE, Jan; VAN GUCHT, Dirk & VANSUMMEREN, Stijn (2007) A Crash Course on Database Queries. In: Libkin, Leonid (Ed.) Proceedings of the Twenty-Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems. p. 143-154..

Handle: http://hdl.handle.net/1942/7736

A Crash Course on Database Queries

Jan Van den Bussche
Hasselt University
transnational University of Limburg

Dirk Van Gucht
Indiana University

Stijn Vansummeren*
Hasselt University
transnational University of Limburg

ABSTRACT

Complex database queries, like programs in general, can 'crash', i.e., can raise runtime errors. We want to avoid crashes without losing expressive power, or we want to correctly predict the absence of crashes. We show how concepts and techniques from programming language theory, notably type systems and reflection, can be adapted to this end. Of course, the specific nature of database queries (as opposed to general programs), also requires some new methods, and raises new questions.

Categories and Subject Descriptors

H.2.3 [**Database Management**]: Languages—*Query Languages*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs; F.4.1 [**Mathematical Logical and Formal Languages**]: Mathematical Logic—*Computability Theory*

General Terms

Verification, Theory, Languages

Keywords

Runtime errors, well-definedness, type systems, typability, type inference, reflection, nested relational calculus, relational algebra, XQuery

1. INTRODUCTION

Research in programming languages has produced sophisticated tools for the analysis and definition of computer programs. The most prominent such tool is the *static type system* whose purpose is to ensure that well-typed programs do not crash [12, 38]. Recall that a program in general has three possible outcomes: it may terminate with a valid result; it may terminate with a runtime error (in which case the program is said to 'crash'); or it may not terminate at all. Runtime errors occur for example when executing instructions like $5 + $ 'John' where the primitive addition operator $+$ is applied to inappropriate values. Although decidable static type systems can prove the absence of crashes, they cannot prove their presence. For example, a program like

if <complex test> then <crash>

will be rejected as ill-typed even if <complex test> never terminates and the <crash> expression is never executed, as termination of programs is undecidable and hence cannot be statically checked.

As a result of this conservatism, the earlier static type systems fell short on flexibility. To paraphrase Milner [34]:

> A widely employed style of programming, particularly in structure-processing languages which impose no discipline of types (LISP is a perfect example), entails defining procedures that work well on objects of a wide variety (e.g., on lists of atoms, integers, or lists). Unfortunately, one often pays a price for this flexibility in the time taken to find rather inscrutable bugs — anyone who mistakenly applies CDR to an atom in LISP, and finds himself absurdly adding a property list to an integer, will know the symptoms. On the other hand, a type discipline such as that of ALGOL 68 which precludes the bugs mentioned above, also precludes the programming style which we are talking about.

The flexibility issue was solved by recognizing that primitive language operators are naturally *polymorphic* and can safely be applied to arguments of a wide range of types. Such operators include assignment, function application, pairing and tupling, and list-processing operators. Independently, Hindley [22] and Milner [15, 34] extended the static type system to deal with polymorphism; they described a *type inference* (sometimes also called *type reconstruction*) algorithm that assigns each program fragment to its *principal type scheme*. This is a concise description of all possible type assignments under which the program is well-typed. For example, the list concatenation function

$$\mathsf{fun}\ \mathrm{concat}(l_1, l_2) = \mathsf{if}\ \mathsf{null}(l_1)\ \mathsf{then}\ \mathsf{nil}\ \mathsf{else}\ \mathsf{cons}(\mathsf{hd}(l_1), \mathrm{concat}(\mathsf{tl}(l_1), l_2))$$

is given the principal type scheme $[\alpha] \times [\alpha] \rightarrow [\alpha]$, which states that concat — being built from polymorphic primitives — is itself polymorphic: it can be applied to any pair of

*Stijn Vansummeren is a Postdoctoral Fellow of the Research Foundation–Flanders (FWO).

PODS'07, June 11–14, 2007, Beijing, China.

lists of objects of the same type and returns a list of objects of that type. This yields the desired flexibility, as programs like

$$(\text{concat}([1,2],[3,4]), \text{concat}(['\text{John}'],['\text{Doe}']),$$

can now be considered well-typed. In contrast, this program is ill-typed in earlier monomorphic type systems where concat can only be given the type $[\mathsf{int}] \times [\mathsf{int}] \rightarrow [\mathsf{int}]$ or $[\mathsf{string}] \times [\mathsf{string}] \rightarrow [\mathsf{string}]$, in which case either concat(['John'],['Doe']) or concat([1,2],[3,4]) is ill-typed.

While the goal of type systems is to guarantee soundness of runs without giving up flexibility in the definition of programs, there is another tool from programming languages, *reflection*, that is squarely directed towards enhancing this flexibility even more. Briefly, reflection is the ability, within the run of a program, to inspect the program, and also generate and execute other programs. So, programs become a kind of data, and one also speaks of meta-programming in this context. The connection between type checking and reflection is not as far-fetched as it may seem, certainly not in the context of semi-structured data models where the distinction between type information and ordinary data is blurred. The additional step from type information to other kinds of program expressions as data is then not so large.

Indeed, in light of the view that database query languages are "just" domain-specific programming languages, the tools of type systems and reflection can of course also be applied in the database context. There are some important differences between general-purpose programming languages and query languages, however, that call for revisiting these tools and studying how they specialize in the database context.

For example, query languages typically are less than turing complete and may therefore exhibit static type systems that are both sound and complete in the sense that they cannot only prove the absence of crashes, but also their presence. Furthermore, one of the prime data structures in databases is the record. Although there has been considerable research in the programming languages community to extend type inference to deal with (polymorphic) record operators, these studies have mainly focused on records in the presence of higher-order function types and subtyping, two features typically not present in query languages. The techniques for and complexity of type inference may therefore differ significantly. Finally, although the unification of programs and data has been put forward as an important database research topic [2], little attention has been given to reflection and meta-programming as a means to this unification.

The goal of this paper is to present a subjective overview of some recent work on types and reflection in the context of query languages. In addition, we highlight some research issues and directions for future work.

2. COMPLETE STATIC TYPE SYSTEMS

Let us first introduce the Nested Relational Calculus $\mathcal{NRC}$, a generalization and elegant abstraction of the familiar select-from-where SQL, OQL, and C$^\sharp$ queries [7, 9]. The $\mathcal{NRC}$ operates on *complex objects* o, which are nested combinations of atomic constants c; records; and sets:

$$o ::= c \mid (A\colon o, \ldots, B\colon o') \mid \{o, \ldots, o'\}.$$

As usual, the attributes in a record $(A\colon o, \ldots, B\colon o')$ are all assumed to be distinct, and the order of attributes does not matter. For example, $(A\colon c, B\colon c') = (B\colon c', A\colon c)$. The $\mathcal{NRC}$ expressions e themselves are given by the syntax

$$\begin{array}{rcl} e & ::= & x \mid o \mid (A\colon e, \ldots, B\colon e') \mid e.A \\ & \mid & \{\} \mid \{e\} \mid e_1 \cup e_2 \mid \{e \mid x_1 \in e_1, \ldots, x_n \in e_n\} \end{array}$$

(where parentheses may be used to avoid ambiguity). Here, x ranges over *variables* that can be bound to input objects; o is constant object formation; $(A\colon e, \ldots, B\colon e')$ is record formation; $e.A$ is field inspection; $\{\}$ and $\{e\}$ are empty and singleton set construction, respectively; $e_1 \cup e_2$ is set union; and $\{e \mid x_1 \in e_1, \ldots, x_n \in e_n\}$ is set comprehension. For example, $\{(A\colon x, B\colon y) \mid x \in R, y \in S\}$ returns the cartesian product of the sets R and S, while $\{y \mid x \in T, y \in x\}$ flattens the set of sets T.

We use the notation Δ as a shorthand for $x_1 \in e_1, \ldots, x_n \in e_n$. It should be emphasized that the $x_i \in e_i$ part in the $\{e \mid \Delta, x_i \in e_i, \Delta'\}$ construct is *not* a membership test. It is an abstraction which introduces and binds the variable x_i, whose scope is the expression e and Δ'. In light of this view, the *free variables* $FV(e)$ of an expression e are hence inductively defined as follows: $FV(x) = \{x\}$, $FV(o) = \{\}$, $FV(\{e \mid x_1 \in e_1, \Delta\} = FV(e_1) \cup (FV(\{e \mid \Delta\}) - \{x\})$, and $FV(e)$ is the union of the free variables of e's immediate subexpressions otherwise. We write $e(x, \ldots, y)$ to indicate that e is an expression with $FV(e) = \{x, \ldots, y\}$). An expression without free variables is *closed*.

Some expressions, like $(C\colon o).A$ and $5 \cup \{o\}$, clearly apply primitive operators to inappropriate objects and will therefore crash during evaluation. This intuition is formalized as follows. Let $e[x/o, \ldots, y/o']$ denote the expression obtained from e by replacing all free occurrences of $x, \ldots, y$ by $o, \ldots, o'$ respectively. Similarly, let $\Delta[x/o, \ldots, y/o']$, with $\Delta = x_1 \in e_1, \ldots, x_n \in e_n$, denote $x_1 \in e_1[x/o, \ldots, y/o'], \ldots, x_n \in e_n[x/o, \ldots, y/o']$. Evaluation of $e(x, \ldots, y)$ on $o, \ldots, o'$ can then be seen as running the operational semantics of Figure 1 on $e[x/o, \ldots, y/o']$. There, we use the notation $e \rightarrow o$ to indicate that *closed* expression e evaluates to object o. Evaluation *crashes* when there is no o such that $e \rightarrow o$.

$$\frac{}{o \rightarrow o} \qquad \frac{e \rightarrow o \quad \ldots \quad e' \rightarrow o'}{(A\colon e, \ldots, B\colon e') \rightarrow (A\colon o, \ldots, B\colon o')}$$

$$\frac{e \rightarrow (A\colon o, \ldots, B\colon o')}{e.A \rightarrow o} \qquad \frac{}{\{\} \rightarrow \{\}} \qquad \frac{e \rightarrow o}{\{e\} \rightarrow \{o\}}$$

$$\frac{e_1 \rightarrow \{o_1, \ldots, o_m\} \quad e_2 \rightarrow \{o'_1, \ldots, o'_n\}}{e_1 \cup e_2 \rightarrow \{o_1, \ldots, o_m, o'_1, \ldots, o'_n\}}$$

$$\frac{e \rightarrow o}{\{e \mid \} \rightarrow \{o\}} \qquad \frac{e_1 \rightarrow \{\}}{\{e \mid x_1 \in e_1, \Delta\} \rightarrow \{\}}$$

$$\frac{\begin{array}{c} e_1 \rightarrow \{o, \ldots, o'\} \\ \{e[x_1/o] \mid \Delta[x_1/o]\} \cup \cdots \cup \{e[x_1/o'] \mid \Delta[x_1/o']\} \rightarrow o'' \end{array}}{\{e \mid x_1 \in e_1, \Delta\} \rightarrow o''}$$

Figure 1: The operational semantics of $\mathcal{NRC}$.

Example 1. Evaluation of the expression $\{(A\colon y.C, B\colon z) \mid y \in x_1, z \in x_2\}$ with x_1 bound to $o_1 = \{(C\colon 1)\}$ and x_2

bound to $o_2 = \{3\}$ is successful:

$$\{(A\colon y.C, B\colon z) \mid y \in \{(C\colon 1)\}, z \in \{3\}\} \rightarrow \{(A\colon 1, B\colon 3)\}.$$

Evaluation of this expression with x_1 bound to $o'_1 = (C\colon 1)$ instead of o_1 crashes, however, as no inference rule applies to $\{(A\colon y.C, B\colon z) \mid y \in (C\colon 1), z \in \{3\}\}$.

Note that crashes only occur when (1) we apply field inspection to a record without the desired field or to a non-record, and (2) when we apply set union and comprehension to non-sets.

We are interested in the crashing behavior of expressions when the inputs are taken from certain prescribed classes of objects. To this end, let the $\mathcal{NRC}$ *types* be given by the syntax

$$s, t ::= \mathsf{atom} \mid (A\colon s, \dots, B\colon t) \mid \{s\},$$

where attribute names occurring in a record type $(A\colon s, \dots, B\colon t)$ are all assumed to be distinct. The semantics of a type is just a set of objects: atom is the set of all atomic data constants (which in practice will include the integers, the strings, and so on); $(A\colon s, \dots, B\colon t)$ is the set of all records $(A\colon o, \dots, B\colon o')$ with $o, \dots, o'$ of type $s, \dots, t$ respectively; and $\{s\}$ is the set of all finite sets of objects of type s. We write $o\colon s$ to indicate that o is an object of type s.

2.1 Well-definedness

The question whether a sound and complete static type system exists for the $\mathcal{NRC}$ is now equivalent to the question whether the following problem is decidable.

Well-Definedness	
Input:	Expression $e(x, \dots, y)$ and types $s, \dots, t$ for the free variables.
Problem:	Decide whether e is *well-defined under* $s, \dots, t$, i.e., whether $e[x/o, \dots, y/o']$ evaluates to an object for all $o\colon s, \dots, o'\colon t$.

It turns out that well-definedness is not only decidable for the $\mathcal{NRC}$ itself but also for $\mathcal{NRC}(\mathsf{eq})$, which is the extension of $\mathcal{NRC}$ with atomic comparison expressions $e_1 \;\mathsf{eq}\; e_2$ and the following evaluation rules, where c_1 and c_2 stand for distinct atoms:

$$\frac{e_1 \rightarrow c_1 \quad e_2 \rightarrow c_1}{e_1 \;\mathsf{eq}\; e_2 \rightarrow \{()\}} \qquad \frac{e_1 \rightarrow c_1 \quad e_2 \rightarrow c_2}{e_1 \;\mathsf{eq}\; e_2 \rightarrow \{\}}$$

Note the classical relational representation of the boolean values: true as the singleton $\{()\}$ containing the empty record, and false as the empty set. This allows queries like 'return all records in R whose A-field is 5' to be expressed as $\{x \mid x \in R, y \in (x.A \;\mathsf{eq}\; 5)\}$.

It should be emphasized that only atomic data values can be compared, as $o_1 \;\mathsf{eq}\; o_2$ crashes when o_1 or o_2 is not an atom. Intuitively, $\mathcal{NRC}(\mathsf{eq})$ is the complex object equivalent of the conjunctive queries [1]. In particular, $\mathcal{NRC}(\mathsf{eq})$ is monotone in the following sense. Define the containment relation $\sqsubseteq$ on objects to be the smallest relation such that $c \sqsubseteq c$ for every atom c; $(A\colon o_1, \dots, B\colon o_n) \sqsubseteq (A\colon o'_1, \dots, B\colon o'_n)$ if $o_i \sqsubseteq o'_i$; and $\{o_1, \dots, o_n\} \sqsubseteq \{o'_1, \dots, o'_m\}$ if every o_i has some o'_j such that $o_i \sqsubseteq o'_j$.

Proposition 2. *$\mathcal{NRC}(\mathsf{eq})$ is monotone with regard to $\sqsubseteq$. That is, for all $\mathcal{NRC}(\mathsf{eq})$ expressions $e(x_1, \dots, x_n)$ and all objects $o_1 \sqsubseteq o'_1, \dots, o_n \sqsubseteq o'_n$, if $e[x_1/o_1, \dots, x_n/o_n] \rightarrow o$ and $e[x_1/o'_1, \dots, x_n/o'_n] \rightarrow o'$, then $o \sqsubseteq o'$. Also, if $e[x_1/o_1, \dots, x_n/o_n]$ crashes, then so does $e[x_1/o'_1, \dots, x_n/o'_n]$.*

Proposition 2 is crucial in showing that $\mathcal{NRC}(\mathsf{eq})$ has the following "small-model property for undefinedness".[1]

Theorem 3 ([55])**.** *Define the width of an object o to be the maximum cardinality of a set occurring in o. Given $e(x, \dots, y)$ in $\mathcal{NRC}(\mathsf{eq})$ and types $s \dots, t$ for the free variables of e it is possible to compute $l \in \mathbb{N}$ such that if e is not well-defined under $s, \dots, t$, there exist $o\colon s, \dots o'\colon t$ of width at most l on which $e[x/o, \dots, y/o']$ crashes.*

Decidability of the well-definedness problem for $\mathcal{NRC}(\mathsf{eq})$ immediately follows. Indeed, up to isomorphism (and expressions cannot distinguish isomorphic inputs), there are only a finite number of objects $o\colon s, \dots, o'\colon t$ of width at most l. It suffices to test them all to see if there is a counter-example to well-definedness.

Theorem 4 ([55])**.** *Well-definedness for $\mathcal{NRC}(\mathsf{eq})$ is decidable.*

Clearly, the method of enumerating all possible counter-examples is computationally quite expensive. To appreciate the inherent complexity of the problem, we note that the classical *satisfiability problem*, defined as

Satisfiability	
Input:	Expression $e(x, \dots, y)$ and types $s, \dots, t$ such that $e[x/o, \dots, y/o']$ evaluates to a set for all $o\colon s, \dots, o'\colon t$.
Problem:	Decide whether there exist $o\colon s, \dots, o'\colon t$ such that $e[x/o, \dots, y/o']$ evaluates to a non-empty set.

reduces to the complement of well-definedness, which we call *ill-definedness*. Indeed, if we assume $e[x/o, \dots, y/o']$ to evaluate to a set for all inputs $o\colon s, \dots, o'\colon t$, then e is satisfiable iff the expression $\{\{\}.A \mid x \in e\}$ is ill-defined under $s, \dots, t$. In particular, when e is closed, deciding ill-definedness of $\{\{\}.A \mid x \in e\}$ is as hard as checking whether e returns a non-empty set. By the results of Koch on the query complexity of closed $\mathcal{NRC}(\mathsf{eq})$ expressions [26] it readily follows that

Theorem 5. *Well-definedness for $\mathcal{NRC}(\mathsf{eq})$ is hard for* Co-Nexptime.

Whether this lower bound is tight remains unknown. The decidability result of Theorem 4 is actually quite sharp. For example, extending the language with general comparison tests $e_1 = e_2$ where

$$\frac{e_1 \rightarrow o_1 \quad e_2 \rightarrow o_1}{e_1 = e_2 \rightarrow \{()\}} \qquad \frac{e_1 \rightarrow o_1 \quad e_2 \rightarrow o_2}{e_1 = e_2 \rightarrow \{\}}$$

turns the problem undecidable. The crucial observation here is that, since $\mathcal{NRC}(=)$ is an extension of the relational algebra [9] for which satisfiability is undecidable [1], satisfiability for $\mathcal{NRC}(=)$ is also undecidable. Hence, by the reduction of satisfiability to ill-definedness it follows that

[1] $\mathcal{NRC}(\mathsf{eq})$ also turns out to have a "small model property for definedness" in correspondence with the small model property of the positive-existential fragment of first order logic [55].

Theorem 6 ([55])**.** *Well-definedness for $\mathcal{NRC}(=)$ is undecidable.*

Another interesting example in point is $\mathcal{NRC}(\mathsf{eq}, \mathsf{extract})$, the extension of $\mathcal{NRC}(\mathsf{eq})$ with the singleton extraction operator $\mathsf{extract}\ e$ where $\mathsf{extract}\ e \rightarrow o$ iff $e \rightarrow \{o\}$. This operator allows us to model the behavior of SQL queries with conditions involving scalar subqueries. Indeed, recall that in SQL the where-clause condition $5 = \mathsf{select\ distinct}\ A\ \mathsf{from}\ R$ crashes if the subquery $\mathsf{select\ distinct}\ A\ \mathsf{from}\ R$ does not return a singleton. This behavior is readily modeled by the expression $5\ \mathsf{eq}\ (\mathsf{extract}\,\{x.A \mid x \in R\})$. We also note that an operator like $\mathsf{extract}$ is explicitly present in OQL [13].

Theorem 7 ([55])**.** *Well-definedness for $\mathcal{NRC}(\mathsf{eq}, \mathsf{extract})$ is undecidable.*

The crucial observation here is that, assuming that e_1 and e_2 are already well-defined, the expression $\mathsf{extract}\ (\{e_1\} \cup \{e_2\})$ is well-defined under $s, \dots, t$ if, and only if, e_1 and e_2 are equivalent. Theorem 7 then follows since, even though $\mathcal{NRC}(\mathsf{eq})$ only expresses *monotone* queries by Proposition 2, equivalence of well-defined $\mathcal{NRC}(\mathsf{eq})$ expressions is undecidable [55].

We should mention that XQuery [6, 16], the standard query language for XML data, has several operators that, like $\mathsf{extract}$, crash on non-singleton inputs. Interestingly enough, well-definedness for fragments of XQuery that include such operators is not necessarily undecidable. The crucial difference with $\mathcal{NRC}(\mathsf{eq}, \mathsf{extract})$ is that XQuery works on trees and lists instead of sets. Equivalence then no longer reduces to well-definedness of $\mathsf{extract}\ (\{e_1\} \cup \{e_2\})$, as the list (e_1, e_2) is never a singleton, whether e_1 and e_2 are equivalent or not.

Let us therefore consider $\mathcal{XQ}(\Omega)$, the non-recursive for-let-where-return data processing fragment of XQuery with primitive operators in Ω. Its expressions α are given by the syntax

$$\begin{array}{lcl}\alpha & ::= & x \mid c \mid () \mid \mathsf{if}\ \alpha\ \mathsf{then}\ \alpha_t\ \mathsf{else}\ \alpha_f \mid \mathsf{let}\ x := \alpha\ \mathsf{return}\ \alpha' \\ & \mid & \mathsf{for}\ x\ \mathsf{in}\ \alpha\ \mathsf{return}\ \alpha' \mid f(\alpha_1, \dots, \alpha_k)\end{array}$$

XQuery expressions manipulate and return *values*, which are finite lists of atomic constants and nodes [6, 17]. Nodes are grouped into a *background store* (a lists of trees) which may be updated during evaluation. The semantics of $\mathcal{XQ}(\Omega)$ is then that x ranges over variables that may be bound to input values; c is atomic constant formation; and $()$ is empty list construction. The conditional $\mathsf{if}\ \alpha\ \mathsf{then}\ \alpha_t\ \mathsf{else}\ \alpha_f$ evaluates α_t when α evaluates to the singleton boolean $[\mathsf{true}]$ and evaluates α_f when α evaluates to $[\mathsf{false}]$. Note in particular that the conditional crashes when α does not evaluate to $[\mathsf{true}]$ or $[\mathsf{false}]$. The let expression $\mathsf{let}\ x := \alpha\ \mathsf{return}\ \alpha'$ evaluates α' with x bound to the result of evaluating α. The for-loop $\mathsf{for}\ x\ \mathsf{in}\ \alpha\ \mathsf{return}\ \alpha'$ evaluates α' for each item x in the result of α and concatenates the resulting values. Finally, $f(\alpha_1, \dots, \alpha_k)$ is primitive operator application (with $f \in \Omega$ and k the arity of f). Examples of such operators include an equality test, the various XPath axes, creating a new tree, and so on. It is important to emphasize that primitive operators in XQuery are *partial* functions. For example, element creation crashes when its first argument is not a singleton list [6].

Although one can study well-definedness of $\mathcal{XQ}(\Omega)$ expressions for each instantiation of Ω separately, there is a general theorem that ensures decidability of the problem when the input types are all given by *bounded depth* regular expression types. Regular expression types are essentially regular tree languages. They naturally occur in XML Schema [50]; are used to describe valid inputs in XQuery; and form the basis of general-purpose programming languages manipulating tree-structured data such as XDuce [23, 24] and $\mathbb{C}$Duce [18]. The bounded-depth restriction is motivated by the observation that most real-world tree-structured data has nesting depth at most five or six [28], and that unbounded-depth nesting is hence often not needed.

Theorem 8 ([57, 58])**.** *If only bounded depth regular expression types are considered, and if Ω contains only operators that are (1) monotone; (2) generic; (3) local; and (4) locally-undefined, then well-definedness for $\mathcal{XQ}(\Omega)$ is decidable.*

Intuitively, monotonicity ensures that satisfiability (which continues to reduces to ill-definedness) is decidable; genericity ensures that we do not run into trouble by interpreting atomic constants; and locality and local-undefinedness ensure that if an expression crashes on some input, it is also crashes on an input whose size can be statically computed from the expression and the input types. All of them taken together ensure that $\mathcal{XQ}(\Omega)$ has a small model property for undefinedness similar to Theorem 3.

Actually, each of monotonicity, genericity, locality, and local-undefinedness are necessary in the sense that omitting any one of them allows for a set of operators that turns well-definedness undecidable [57, 58].

Decidability of well-definedness for a large and practical fragment of XQuery immediately follows from Theorem 8 as, in the absence of automatic coercions, the various XPath axes; node constructors; value and node comparisons; and node label and content inspections are monotone, generic, local, and locally-undefined [57, 58]. Since satisfiability continues to reduce to ill-definedness of closed expressions, the results of Koch [26] imply that well-definedness for this fragment is CO-NEXPTIME hard. As for $\mathcal{NRC}(\mathsf{eq})$, it is unknown whether this lower bound is tight.

Another open question is whether the bounded depth restriction on the regular expression types can be relaxed.

2.2 Semantic type-checking

The question whether sound and complete static type systems exist for database query languages can also be viewed from a different angle. A useful side-effect of type systems is that they can also be used to verify that all outputs of a program belong to a certain output type. This is especially useful in semi-structured databases, where data produced by a query is often expected to adhere to a prescribed type. Again, type systems for general purpose programming languages can prove that all outputs are of the desired type, but cannot prove that some output is not. Viewed from this angle, the question whether a sound and complete static type system exits corresponds to decidability of the following problem.

SEMANTIC TYPE-CHECKING

Input: Expression $e(x, \dots, y)$, well-defined under $s, \dots, t$ and an additional type r.

Problem: Decide whether $e[x/o, \dots, y/o']$ evaluates to an object in r, for all $o\colon s, \dots, o'\colon t$.

$$\frac{}{T \vdash x\colon T(x)} \qquad \frac{o\colon s}{T \vdash o\colon s} \qquad \frac{T \vdash e\colon (A\colon s, \ldots, B\colon t)}{T \vdash e.A\colon s}$$

$$\frac{T \vdash e\colon s \quad \ldots \quad T \vdash e'\colon t}{T \vdash (A\colon e, \ldots, B\colon e')\colon (A\colon s, \ldots, B\colon t)}$$

$$\frac{}{T \vdash \{\}\colon \{s\}} \qquad \frac{T \vdash e\colon s}{T \vdash \{e\}\colon \{s\}} \qquad \frac{T \vdash e_1\colon \{s\} \quad T \vdash e_2\colon \{s\}}{T \vdash e_1 \cup e_2\colon \{s\}}$$

$$\frac{\begin{array}{c} x_1\colon s_1, \ldots, x_i\colon s_i, T \vdash e_{i+1}\colon \{s_{i+1}\} \text{ for } 0 \leq i < n \\ x_1\colon s_1, \ldots, x_n\colon s_n, T \vdash e\colon s \end{array}}{T \vdash \{e \mid x_1 \in e_1, \ldots, x_n \in e_n\}\colon \{s\}}$$

$$\frac{T \vdash e_1\colon s \quad T \vdash e_2\colon s}{T \vdash e_1 = e_2\colon \{()\}}$$

Figure 2: Static type system of $\mathcal{NRC}(=)$.

In the XML-related setting where expressions can only inspect and manipulate a fixed, *finite* alphabet of atomic constants (serving as tree labels), semantic type-checking can be realized by a reduction to the satisfiability problem of monadic second-order logic over trees, which is known to be decidable [49]. Of course, the complexity of the problem varies widely depending on both the expressiveness of the language considered and the class of input and output types allowed [29, 30, 31, 32, 35, 45, 62]. In contrast, when an *infinite* set of atomic constants is allowed, the problem quickly becomes undecidable [3, 4]. In the presence of bounded-depth regular expression types, this is even true for monotone languages. In contrast, for $\mathcal{NRC}(eq)$ we have:

Theorem 9 ([55]). *Semantic type-checking for $\mathcal{NRC}(\mathsf{eq})$ is decidable.*

3. INCOMPLETE STATIC TYPE SYSTEMS

From Section 2 we may conclude that we can check crashes in a sound and complete way for the (restricted yet useful) class of monotone queries. The CO-NEXPTIME hardness results make it unlikely, however, that these checks can be made practical. Furthermore, for non-monotone queries we must always revert to a traditional, incomplete (but efficient) type system.

These observations call for a closer inspection of the traditional type systems used in query languages. A particular example in point is the type system for $\mathcal{NRC}(=)$, given in Figure 2. There, T stands for a *type assignment* (a mapping from variables to types) and the notation $x\colon s, T$ stands for the type assignment that equals T on all variables except x, which it maps to s. As usual, the notation $T \vdash e\colon s$ indicating that *e has type s under T* should be read as "assuming that the free variables x of e are bound to objects of type $T(x)$, e outputs objects of type s". Observe that this relation only depends on the free variables of an expression: if T and T' agree on $FV(e)$ and $T \vdash e\colon s$, then also $T' \vdash e\colon s$. We may therefore write $x\colon r, \ldots, y\colon s \vdash e\colon t$ as a shorthand of the more verbose $x\colon r, \ldots, y\colon s, T \vdash e\colon t$ when $FV(e) = \{x, \ldots, y\}$.

We adopt the convention that the order of attributes in a record type is irrelevant and that hence $(A\colon s, B\colon t) = (B\colon t, A\colon s)$. As such, the typing rule for field inspection states that $e.A$ has type s under T whenever e has a record type whose A attribute is of type s, not only when A happens to be the first attribute mentioned in that record type.

The obvious property one expects from a type system is *soundness*:

Theorem 10. *The static type system of Figure 2 is sound. That is, if $FV(e) = \{x, \ldots, y\}$ and $x\colon r, \ldots, y\colon s \vdash e\colon t$ then for all $o\colon r, \ldots, o'\colon s$ there exists $o''\colon t$ such that $e[x/o, \ldots, y/o'] \to o''$.*

Note that soundness implies well-definedness. The converse implication does not hold however, as the static type system is not *complete*. For example, $\{\{\}.A \mid x \in \{\}\}$ is well-defined, but is not *well-typed* (i.e., there is no s such that $\vdash e\colon s$).

3.1 Expressive completeness

We should emphasize that devising a type system that only needs to be sound is trivial. It suffices to let every expression be ill-typed no matter the type assignment, as soundness vacuously holds in the absence of well-typed expressions.

Of course, such a type system is useless as it precludes the definition of *all* queries that can be expressed in a well-defined (but untyped) manner. Although the $\mathcal{NRC}(=)$ type system from Figure 2 is far from trivial, the question of its expressive power with regard to the class of well-defined queries remains. Observe, for example, that well-defined expressions may manipulate heterogeneous sets (i.e., sets of objects of different types), while well-typed expressions cannot.

Example 11. The expression $e = \{z.A \mid z \in (x \cup y)\}$ is well-defined under $x\colon \{(A\colon s, B\colon s)\}, y\colon \{(A\colon r, C\colon t)\}$. It is not well-typed under this type assignment, however, as the type rule for $x \cup y$ requires x and y to have the same set type. Nevertheless, the same query is expressed by $e' = \{z.A \mid z \in x\} \cup \{z.A \mid z \in y\}$, which is well-typed.

Whether this example can be generalized to all well-defined expressions is still unknown. We strongly conjecture that it can, however.

Conjecture 12. *The static type system from Figure 2 is expressively complete. That is, every $\mathcal{NRC}(=)$ expression $e(x, \ldots, y)$ that is well-defined under $r, \ldots, s$ and only produces outputs in a type t has an equivalent expression $e'(x, \ldots, y)$ such that $x\colon r, \ldots, y\colon s \vdash e'\colon t$.*

Most type systems for turing complete programming languages are easily shown expressively complete: it suffices to show that one can simulate all turing machine operations (including encoding and decoding of the programming language objects on turing machine tapes) in a well-typed manner. Proving Conjecture 12, in contrast, is more difficult exactly because $\mathcal{NRC}(=)$ is not turing complete.

Interestingly enough, there are also type systems for turing complete programming languages that are *not* expressively complete. For example, the untyped lambda calculus can define all computable functions, while in the simply typed lambda calculus only a restricted class of functions, the so-called *extended polynomials*, are definable [5, 42].

$$\frac{T \vdash e\colon (A\colon r, B\colon s, \ldots, C\colon t)}{T \vdash \mathsf{drop}_A\ e\colon (B\colon s, \ldots, C\colon t)}$$

$$\frac{T \vdash e_1\colon \{\phi_1\} \quad T \vdash e_2\colon \{\phi_2\} \quad \phi_1 \text{ and } \phi_2 \text{ have disjoint sets of attributes}}{T \vdash e_1 \times e_2\colon \{\phi_1 + \phi_2\}}$$

$$\frac{T \vdash e_1\colon \{\phi_1 + \psi\} \quad T \vdash e_2\colon \{\phi_2 + \psi\} \quad \phi_1 \text{ and } \phi_2 \text{ have disjoint sets of attributes}}{T \vdash e_1 \bowtie e_2\colon \{\phi_1 + \phi_2 + \psi\}}$$

Figure 3: Additional type rules for $\mathcal{NRC}(=, \mathsf{drop})$.

3.2 Polymorphism

The basic operators of the statically typed $\mathcal{NRC}(=)$ are inherently *polymorphic*: we can inspect the A attribute of any record, as long as it has such an A attribute; we can take the union of any two sets of the same type; and we can iterate over any set, no matter the type of its elements. Hence expressions, being built from polymorphic operators, are themselves polymorphic. In particular, the *same* expression can be used to operate on objects of a wide variety of types. For example, the expression $e = \{z.A \mid z \in (x \cup y)\}$ is well-typed under all type assignments mapping x and y to $\{s\}$ with s a record type containing attribute A.

Polymorphism is tied to the familiar database principle of "logical data independence". By this principle, a query formulated on the logical level must not only be insensitive to changes on the physical level, but also to changes to the database schema (i.e., the type), as long as these changes are to parts of the schema on which the query does not depend. This is apparent for e above, as it is still well-typed if we drop from s some attribute B different from A. In other words, polymorphism gives us the flexibility to query different data sources with distinct schemas using the same query, hence enabling code reuse.

Of course, the polymorphism provided by $\mathcal{NRC}(=)$ has its limits. For example, the query that drops attribute A from record x is readily expressed by $(B\colon x.B)$ if we fix the type of x to be $(A\colon r, B\colon s)$, but there intuitively does not seem to be a single expression defining the query that is well-typed whenever x has attribute A. Indeed, the above expression becomes ill-typed for x of type $A\colon r$ and is incorrect for x of type $(A\colon r, B\colon s, C\colon t)$.

To make this intuition rigorous, let us call a pair (T, s) such that $T \vdash e\colon s$ a *typing* of e. Two expressions $e_1(x, \ldots, y)$ and $e_2(x, \ldots, y)$ are *polymorphically equivalent* if they have the same set of typings and, for each such typing (T, s), e_1 and e_2 evaluate to the same object on all inputs $o\colon T(x)$, $\ldots$, $o'\colon T(y)$. Also, let $\mathcal{NRC}(=, \mathsf{drop})$ be the extension of $\mathcal{NRC}(=)$ with expressions of the form $\mathsf{drop}_A\ e$ capable of dropping attribute A:

$$\frac{e \to (A\colon o, B\colon o', \ldots, C\colon o'')}{\mathsf{drop}_A\ e \to (B\colon o', \ldots, C\colon o'')}$$

The corresponding type rule is given in Figure 3.

Theorem 13. *No expression in $\mathcal{NRC}(=)$ is polymorphically equivalent to $\{\mathsf{drop}_A\ x \mid x \in R\}$. Consequently, no expression in $\mathcal{NRC}(=)$ is polymorphically equivalent to $\mathsf{drop}_A\ x$.*

This result (which we will prove in Section 3.3) was first obtained for the relational algebra [54]. In fact, many classical operators that are "derived" in the standard relational algebra become primitive in the polymorphic setting. For example, the semijoin operator $\ltimes$ has no polymorphic equivalent in the relational algebra, while cartesian product $\times$ and join $\bowtie$ are polymorphically independent of each other [54]. To illustrate the latter, observe that the simulation of $R \bowtie S$ by means of projection, selection, renaming, and cartesian product $\pi(\sigma(\rho(R) \times \rho(S)))$ only works if we know the schemas of R and S, otherwise we do not know what renamings to apply. Conversely, although $R \times S$ is equivalent to $R \bowtie S$ when R and S have disjoint sets of attributes, $R \bowtie S$ is well-typed whenever R and S agree on the types of the attributes they have in common, whereas $R \times S$ is well-typed only when R and S have no attributes in common.

The situation in the presence of complex objects is similar. Let $\mathcal{NRC}(=, \mathsf{drop}, \times, \bowtie)$ be the extension of $\mathcal{NRC}(=, \mathsf{drop})$ with expressions of the form $e_1 \times e_2$ and $e_1 \bowtie e_2$ that perform the cartesian product and natural join of two sets of records, respectively. The corresponding typing rules are given in Figure 3, where ϕ_1, ϕ_2, and ψ stand for record types like $(A\colon r, B\colon s, C\colon t)$ and $\phi_1 + \phi_2$ stands for the *extension of ϕ_1 by attributes in ϕ_2*. This is the record type we obtain by adding to ϕ_1 all attributes and types in ϕ_2 that do not occur in ϕ_1. For example, $(A\colon r, B\colon s, C\colon t) + (B\colon t, D\colon r) = (A\colon r, B\colon s, C\colon t, D\colon r)$. As expected, the type rule for $e_1 \times e_2$ states that if e_1 and e_2 are sets of records with disjoint attributes, then the result is a set of records with all these attributes. Similarly, the type rule for $e_1 \bowtie e_2$ states that if e_1 and e_2 are sets of records that agree on the types of their common attributes (given by ψ), then the output type is a set of records containing all attributes of e_1 and e_2.

Theorem 14. *No expression in $\mathcal{NRC}(=, \mathsf{drop}, \times)$ is polymorphically equivalent to $e_1 \bowtie e_2$. Similarly, no expression in $\mathcal{NRC}(=, \mathsf{drop}, \bowtie)$ is polymorphically equivalent to $e_1 \times e_2$.*

One can come up with many more operators that are polymorphically inexpressible even in $\mathcal{NRC}(=, \mathsf{drop}, \times, \bowtie)$. An interesting question for further research is therefore what operators yield a language that is in some sense "polymorphically complete"? In the extreme, the answer is already given by query languages for semi-structured data models (such as XML) that incorporate type information in the data itself, thereby allowing type inspection during evaluation. Query languages for these models are therefore essentially type-independent and extremely polymorphic. But perhaps far less drastic measures are needed to reach the same degree of polymorphism.

3.3 Type Inference

The polymorphic nature of expressions as introduced in Section 3.2 immediately raises the question of *how* one can compute the set of all type assignments under which a given expression it is well-typed. Such *type inference* is often useful in practice. For example, systems such as Kleisli [63] query highly heterogeneous and remote data sources, ranging from traditional relational databases to non-traditional complex structured files to data generated by specialized software packages. While some of these sources have schemas that are accessible, many lack them. Type inference can be helpful in telling for which kinds of sources a given query is suitable, and is in fact imperative for query optimization [63]. Moreover, even though query languages for semi-structured data models are essentially schema-independent, querying is more effective if at least some form of schema is available (perhaps computed from the particular instance) [8, 20]. Although it has received little attention in this context, type inference can then be helpful in telling for which schemas a given query is suitable. Also, *stored procedures* [33] are 4GL and SQL code fragments stored in database dictionary tables. Whenever the schema changes, some of the stored procedures may become ill-typed, while others that were ill-typed may become well-typed. Having an explicit logical description of all typings of each stored procedure may be helpful in this regard.

Another motivation for type inference stems from the area of database programming languages. Recall that a database programming language is a general-purpose programming language featuring a native, integrated query language. Recent examples of such languages include of course XQuery [6], but also C$^\sharp$ and Visual Basic [7]. As type annotations are often not required for expressions of the integrated query language, type inference for such expressions forms a cornerstone of the type checking algorithm of the entire language. In particular, type inference must identify *untypable* expressions that have no typings, like $\{\}.A$ and $x.A \cup x$, as these are ill-typed no matter the context in which they are used.

Inferring types for a given language requires two ingredients: (1) a notion of *type formulas* capable of describing the set of all typings of expressions in the language; and (2) an algorithm that is capable of effectively computing such formulas starting from the expressions themselves. For $\mathcal{NRC}(=)$, the type formulas are constructed from the *polytypes* π, which are the extension of ordinary types with type variables α, as given by the syntax

$$\pi ::= \alpha \mid \mathsf{atom} \mid (A\colon \pi, \ldots, B\colon \pi') \mid \{\pi\}.$$

The semantics of a polytype π is just a set of types. In particular, it is the set $\{\sigma(\pi) \mid \sigma \text{ a substitution}\}$, where a *substitution* is a mapping from type variables to types, and where $\sigma(\pi)$ stands for the type obtained from π by replacing all type variables α by $\sigma(\alpha)$. For example, the semantics of $\{\alpha\}$ is the set $\{\{\mathsf{atom}\}, \{\{\mathsf{atom}\}\}, \{(A\colon \mathsf{atom}, B\colon \mathsf{atom})\}, \ldots\}$.

Polytypes alone do not suffice to describe the set of all typings of a given expression. Rather, we also need the concept of a *kinding assignment*, which is a mapping κ from a finite set of type variables to record polytypes $(A\colon \pi, \ldots, B\colon \pi')$. A substitution σ *respects* κ if $\kappa(\alpha) = (A\colon \pi, \ldots, B\colon \pi')$ implies that $\sigma(\alpha) = (A\colon \sigma(\pi), \ldots, B\colon \sigma(\pi')) + r$ for some record type r.[2] For example, if $\kappa(\alpha) = (A\colon \alpha')$ and α' is not in the domain of κ, then any σ that maps α to a record type with attribute A respects κ.

Type inference for $\mathcal{NRC}(=)$ is a particular instance of type inference for the database programming language Machiavelli [10] as studied by Ohori and Buneman. Their results imply:

Theorem 15. *There exists a polynomial time algorithm that, given an expression $e(x, \ldots, y)$ in $\mathcal{NRC}(=)$, returns* false *if e is untypable, and otherwise returns a formula of the form $\kappa; x\colon \pi_x, \ldots, y\colon \pi_y \to \pi$ such that*

$$\{(T, \sigma(\pi)) \mid \sigma \text{ a substitution respecting } \kappa \text{ and } T(x) = \sigma(\pi_x), \ldots, T(y) = \sigma(\pi_y)\}.$$

is exactly the set of e's typings.

For example, when $e = \{x.A \mid x \in R\}$ this algorithm returns $\kappa, R\colon \{\alpha\} \to \{\alpha'\})$ with $dom(\kappa) = \{\alpha\}$ and $\kappa(\alpha) = (A\colon \alpha')$. It returns false on the untypable $\{\{\}.A \mid x \in \{\}\}$ and $x.A \cup x$.

Theorem 15 actually implies Theorem 13:

Proof of Theorem 13. Suppose, for the purpose of contradiction, that some expression e in $\mathcal{NRC}(=)$ is polymorphically equivalent to $\{\mathsf{drop}_A\ x \mid x \in R\}$. In particular, e and $\{\mathsf{drop}_A\ x \mid x \in R\}$ have the same non-empty set of typings which, by application of Theorem 15 on e, is described by some $\kappa; R\colon \pi_R \to \pi$. Since (T, s) can be a typing of $\{\mathsf{drop}_A\ x \mid x \in R\}$ only if $s = \{t\}$ with t a record type not containing attribute A, π must be of the form $\{(B\colon \pi_B, \ldots, C\colon \pi_C)\}$ for some attributes $B, \ldots, C$. (If π is of the form $\{\alpha\}$ with $\kappa(\alpha) = (B\colon \pi_B, \ldots, C\colon \pi_C)$ then we can always instantiate π to $\{t\}$ with t a record type containing A.) But now the typing $(T, \{(D\colon s\})\})$ with $D \notin \{A, B, \ldots, C\}$ and $T(R) = \{(A\colon r, D\colon s)\}$ is not described by $\kappa; R\colon \pi_R \to \pi$ although it is a typing of $\{\mathsf{drop}_A\ x \mid x \in R\}$. This gives the desired contradiction. $\square$

In other words, the formulas $\kappa; x\colon \pi_x, \ldots, y\colon \pi_y \to \pi$ are unsuitable to describe the typings of $\mathcal{NRC}(=, \mathsf{drop})$ expressions because kinding assignments can only require that some attributes are present in a record type, not that they are absent. This can be resolved by moving to *type schemes* τ which in addition to type variables also contain row variables ρ:

$$\tau ::= \alpha \mid \mathsf{atom} \mid (A\colon \tau, \ldots, B\colon \tau') \mid (A\colon \tau, \ldots, B\colon \tau'; \rho) \mid \{\tau\}.$$

Each row variable comes with a fixed finite set of attributes $\mathrm{attr}(\rho)$. The semantics of a polytype τ is again a set of ordinary types and is defined as follows. First, extend the notion of a substitution to be a function that maps type variables to types and row variables to record types such that $\sigma(\rho)$ contains *no* attributes in $\mathrm{attr}(\rho)$. Then extend substitutions to type schemes as follows:

$$\begin{aligned} \sigma(\mathsf{atom}) &= \mathsf{atom} \\ \sigma(A\colon \tau, \ldots, B\colon \tau') &= (A\colon \sigma(\tau), \ldots, B\colon \sigma(\tau')) \\ \sigma(A\colon \tau, \ldots, B\colon \tau'; \rho) &= (A\colon \sigma(\tau), \ldots, B\colon \sigma(\tau')) + \sigma(\rho) \\ \sigma(\{\tau\}) &= \{\sigma(\tau)\}. \end{aligned}$$

The semantics of a type scheme τ is then the set $\{\sigma(\tau) \mid \sigma \text{ a substitution}\}$.

The techniques of Rémy [39] can then be used to show that:

Theorem 16. *There exists a polynomial time algorithm that, given an expression $e(x, \ldots, y)$ in $\mathcal{NRC}(=, \mathsf{drop})$, returns* false *if e is untypable, and otherwise returns a formula $x\colon \tau_x, \ldots, y\colon \tau_y \to \tau$ such that*

$$\{(T, \sigma(\tau)) \mid \sigma \text{ a substitution and } T(x) = \sigma(\tau_x), \ldots, T(y) = \sigma(\tau_y)\}.$$

is exactly the set of e's typings.

[2] Recall from Section 3.2 that $+$ stands for the extension of record types.

For example, when $e = \{x.A \mid x \in R\}$ this algorithm returns $R\colon \{(A\colon \alpha; \rho)\} \to \{\alpha\}$ with $\mathrm{attr}(\rho) = \{\}$. When $e = \{\mathsf{drop}_A\ x \mid x \in R\}$ it returns $R\colon \{(A\colon s; \rho)\} \to \{\rho\}$ with $\mathrm{attr}(\rho) = \{A\}$.

Using a similar reasoning to the proof of Theorem 13, one can show that the formulas with type schemes as above are unsuitable to describe the typings of expressions like $x \times y$ and $x \bowtie y$. One possible remedy to this problem is to allow record type schemes with *multiple* row variables like $(A\colon r; \rho, \rho')$. Substitutions operate on such schemes in the obvious way:

$$\begin{aligned}&\sigma(A\colon \tau, \dots, B\colon \tau'; \rho, \dots, \rho')\\ &\qquad = (A\colon \sigma(\tau), \dots, B\colon \sigma(\tau')) + \sigma(\rho) + \dots + \sigma(\rho').\end{aligned}$$

If we adopt the convention that distinct row variables can only be substituted with record types having disjoint set of attributes, then the formula $x\colon \{(\rho)\}, y\colon \{(\rho')\} \to \{(\rho, \rho')\}$ faithfully describes the typings of $x \times y$. Also, the formula $x\colon \{(\rho, \rho')\}, y\colon \{(\rho', \rho'')\} \to \{(\rho, \rho', \rho'')\}$ describes the typings of $x \bowtie y$. This approach lies at the basis of a type inference algorithm for the relational algebra [54]. It has the disadvantage that for expressions like

$$R_1 \bowtie (R_2 \bowtie (\cdots \bowtie R_n) \dots),$$

the inferred type formula needs one row variable for each subset $\{i, \dots, j\} \subseteq \{1, \dots, n\}$ to describe the attributes that only inputs $R_i, \dots, R_j$ have in common. As such, the inferred type formulas can be of exponential size.

In the theory of programming languages one also finds type inference algorithms for languages with operators like $\times$ and $\bowtie$, often in the presence of even more powerful features such as higher order functions [10, 37, 46, 47, 61]. There, the preferred solution for describing the typings of $x \times y$ and $x \bowtie y$ is to move to *constrained type formulas*. These are formulas of the form $C; x\colon \tau_x, \dots, y\colon \tau_y \to \tau$ where C is often a conjunctive logical formula that constrains the legal substitutions of the row variables occurring in $\tau_x, \dots, \tau_y, \tau$. For example, if $\rho \# \rho'$ denotes that ρ and ρ' should be substituted with record types having disjoint sets of attributes and if $\rho = \rho' + \rho''$ denotes that ρ should only be substituted with the extension of ρ' and ρ'', then the typings of $x \times y$ is described by

$$\rho' \# \rho'' \wedge \rho = \rho' + \rho'';\ x\colon \{\rho'\}, y\colon \{\rho''\} \to \{\rho\}.$$

This approach can be followed to do type inference for the full $\mathcal{NRC}(=, \mathsf{drop}, \times, \bowtie)$ [56]. It has the advantage that, in contrast to the type inference algorithm for the relational algebra [54], a type formula for a given $\mathcal{NRC}(=, \mathsf{drop}, \times, \bowtie)$ expression can always be inferred in polynomial time [56]. It has the disadvantage that, in contrast to the type inference algorithms for the relational algebra and $\mathcal{NRC}(=, \mathsf{drop})$, the inferred constrained type formulas may become quite complex (which makes them less suitable for presentation to the user) and may even be unsatisfiable. In particular, the constraint-based type inference algorithm for $\mathcal{NRC}(=, \mathsf{drop}, \times, \bowtie)$ returns an unsatisfiable type formula instead of false on untypable expressions.

We should emphasize that it is unlikely that any type inference algorithm for the relational algebra or $\mathcal{NRC}(=, \mathsf{drop}, \times, \bowtie)$ that outputs false on untypable expressions runs in less than exponential time. Indeed, already the typability problem, defined as

TYPABILITY	
Input:	Expression $e(x, \dots, y)$.
Problem:	Decide whether $T \vdash e\colon s$ for some (T, s).

is NP-complete for both the relational algebra [54, 59] and $\mathcal{NRC}(=, \mathsf{drop}, \times, \bowtie)$ [56]. Notice that in contrast, typability for $\mathcal{NRC}(=, \mathsf{drop})$ is in polynomial time by Theorem 16.

4. REFLECTION

At the end of Section 3.2, we already referred to the situation in XML, where we are able to do type checking at run time in the language itself: we can, within an XQuery program, check whether some value is of some XML Schema type. Such languages are said to be capable of *type reflection*. Type reflection is possible in many languages, and depending on the underlying data model and type system, it comes very naturally (such as in XQuery but also much earlier in Scheme) or it must really be provided as an extra feature (such as the reflection package in Java but also much earlier the metaclasses in Smalltalk).

In the context of the relational data model or the complex-object data model that we have been considering in this paper, type reflection as an extra feature of query languages has been studied under the heading "schema querying" [14, 27, 40]. Moreover, data models and query languages have been designed in which not only schema values, such as attribute names or relation names, can be made available as data values, but also vice versa: data values can be "promoted" to schema values [21, 25, 64].

The concept of reflection goes further than mere type reflection, however. When programs, or program fragments, can be treated as values, which can also be generated, inspected, and interpreted at run time, we obtain a more general kind of reflection. This kind of reflection is as old as the concept of universal turing machine, which takes an arbitrary turing machine M represented as a string, as input, and runs M on the fly. Likewise, in the Scheme language, there is an explicit built-in function `eval` that takes a program, represented as a nested list, and runs P on the fly. Of course, as shown by the universal turing machine, in computationally complete languages (like Scheme), such a function `eval` is not a primitive, but merely a convenient feature to allow for a more natural or succinct expression of certain advanced programming constructions. (Closer to home, one can easily imagine a Java interpreter written in Java itself.)

The situation changes, however, when dealing with query languages that are typically not computationally complete. For concreteness, consider XQuery expressions (XQuery without function definitions, so that it is not computationally complete). We can naturally represent the syntax tree of an XQuery expression in XML, as done for example in XQueryX [65]. It thus becomes natural to enhance XQuery with `eval`, and wonder whether `eval` is really primitive: can we write an XQuery expression interpreter using an XQuery expression? Intuitively the answer is negative, and this question has been formally studied in the context of the relational algebra [51].

Concretely, suppose we fix a relational database schema, and agree about some standard way to represent the syntax tree of a relational algebra expression in one or more relations. There are many natural ways to do this; it is only important that this is done in such a way that a total order on the components of the expression can be recovered using the relational algebra. This can be accomplished, e.g., by including the descendant relation of the syntax tree. Then the new operator $\texttt{eval}(r, \dots, s)$ evaluates the relational algebra expression stored in the relations $r, \dots, s$. What happens with the expressive power of the relational algebra when we add in this new operator? Obviously it goes up drastically, because the complexity of evaluating an arbitrary relational algebra expression is PSPACE-complete [60], so the data complexity of `eval` is PSPACE-complete, whereas the data complexity of the relational algebra is in LOGSPACE [1].

The question becomes more interesting, however, when we consider the expressive power of relational algebra with `eval` with respect to standard generic queries: the input is a normal relational database containing no stored expressions. Then, the dynamic generation and evaluation, during a query, of expressions that can depend on the input, becomes a purely computational tool, which now indeed adds expressive power to the relational algebra:

Theorem 17 ([51])**.** *A generic query is expressible in the relational algebra enhanced with* `eval`*, if and only if it is expressible in the relational algebra enhanced with for-loops.*

Similarly, when recursive reflection is considered, where expressions evaluated using `eval` may contain `eval` operators in turn, we get a more powerful equivalence with while-loops instead of for-loops.

Example 18. Let us give an example of the power of reflection in the context of XQuery. Consider an XML document D with the following structure:

$$R \to T^* \qquad T \to A, B$$
$$A \to \#\texttt{PCDATA} \qquad B \to \#\texttt{PCDATA}$$

Such a document represents a binary relation, and the task is to compute the transitive closure of this relation. This is impossible with a single XQuery expression, but using reflection, we can do it as follows. Let n be the number of T-elements in D; in XPath, n equals $\texttt{count}(\$D//T)$. For each $j \in \{1, \dots, n\}$, consider the following expression E_j:

for t_1 in D//T, ..., t_j in D//T return
if every z in ((t_1/B=t_2/A),...,(t_{j-1}/B=t_j/A))
satisfies z=fn:true() then
element(T){t_1/A,t_j/B} else ()

The concatenation expression E of $E_1, \dots, E_n$ clearly computes the transitive closure of D. Note that the syntax tree of E has bounded depth: the j different for-assignments in E_j are j children of one FLRW node; the $j-1$ different equality tests in E_j are j children of one sequence construction node; and likewise, the n different subexpressions E_j of E are n children of one sequence construction node. In particular, an XML representation of E's syntax tree can be constructed from D by a single XQuery expression F. We leave the writing of F as an exercise to the reader; note that the only way in which E depends on D is through n. Then $\texttt{eval}(F)$ is a program that computes the transitive closure of D.

4.1 Reflection and typing

So far, we have been considering untyped reflection, meaning that `eval` can be applied to any subexpression e. Only at run time it is checked that e indeed evaluates to a value that represents a legal expression; if it does not, `eval` will crash. Moreover, even if it does not crash, we do not know what will be the type of the result of the evaluation.

The same problem already occurs with run-time type reflection. For example, assume that t is a record variable holding a record without a field A, and that x is a field variable that has value A; then the evaluation of $t.x$ will crash. Most approaches to schema query languages avoid such crashes by masking them by the boolean value 'false'. More concretely, one turns all operations into predicates: one needs to use a value variable v, and evaluate the predicate $t.x = v$. When t has no field x, the predicate will simply evaluate to false. This is not always very satisfactory, because the predicate will also evaluate to false when t does have a field x, but the field's value is different from v. Clearly one wants to distinguish these two very different origins of the value false.

Sheard and his collaborators have shown how reflection can be typed in the context of the MetaML language [48, 11]. In the same vein, we can define a typed reflective extension of the relational algebra [36]. The three basic ideas are the following: (1) in relations, distinguish between data attributes that store ordinary atomic data values, and expression attributes that store relational algebra expressions; (2) expression attributes are typed by relation schemes: all expression stored in the column of an attribute typed by relation scheme S evaluate to relations of scheme S; (3) provide special operators to syntactically manipulate stored expressions: these operators are strongly typed so that the relation scheme of the resulting expressions is determined by the relation schemes of the input expressions. Using these ideas, one can design a statically typed reflective extension of the relational algebra, where well-typed expressions, including `eval`, will never crash.

It appears that soundness has a price though. First, in the approach just mentioned [36], the reflective relational algebra has not more expressive power than the standard relational algebra without `eval`, as far as standard generic database queries are concerned. This is in sharp contrast with Theorem 17. Second, in a typed meta-programming language, it is not easy to attain sufficient expressive power in the syntactic manipulation of stored expressions. For example, pattern matching is a very useful syntactic operation, but expressions of wildly varying output types can match the same pattern. The design of compile-time reflective query languages is certainly not yet a closed area.

4.2 Reflection in SQL

Using modern SQL/XML technology, it is very easy to implement a reflective extension of SQL [52, 53]. Expressions, in XML format using an appropriate DTD for SQL expressions, can be stored in XML columns of tables. Using the SQL/XML functions `XMLQUERY`, `XMLTABLE`, and `XMLEXISTS`, stored expressions can be syntactically manipulated and queried. So it suffices to add a table-valued user-defined function `SQLEVAL` which takes as input an XML value representing an SQL expression; submits this query to the database system; and returns the resulting table. It is a student exercise to write such a function `SQLEVAL` in Java using JDBC and some appropriate XML parser.

Example 19. To give an idea of the possibilities of reflective SQL, consider a table $T(N, V)$ storing a set of views: N of type string is the name of the view, and V of type XML is the view expression. Suppose we know that each view stored in T has a column `id`. Now we want to know the names of the views that contain the id 345 when evaluated on the current database instance. For that we could write:

```
select N
from T, table(SQLEVAL(T.V)) as S(id)
where S.id=345
```

For another example, suppose the underlying database has two ordinary tables R1 and R2, and we want to know the names of the views stored in T that would show up new id's if we added all tuples of R2 to R1. For that, we need an XQuery function `my:replace` that replaces, in a given XML document representing an SQL expression, every table reference to R1 by the subquery `(table R1 union table R2)`. (Actually such a replace function is much easier to write in XSLT than in XQuery.) We can then write in reflective SQL:

```
select N
from T, table(SQLEVAL(T.V)) as S(id)
where id not in table(SQLEVAL(my:replace(T.V)))
```

We should not forget that the idea of stored query expressions, and their dynamic evaluation in other queries, was already proposed by Stonebraker in 1984 [43, 44]. We also note that a limited form of reflection in SQL (the reflection is essentially limited to where-clauses), is already supported by Oracle [19].

Acknowledgment

Jan Van den Bussche is grateful to Emmanuel Waller, who first introduced him to polymorphic type checking more than ten years ago. Dirk Van Gucht is grateful to Patrick C. Fischer and Daniel Friedman who introduced him to the nested relational data model (more than twenty years ago) and to reflection in programming languages (more than fifteen years ago), respectively.

5. REFERENCES

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

[2] Serge Abiteboul, Rakesh Agrawal, Phil Bernstein, Mike Carey, Stefano Ceri, Bruce Croft, David DeWitt, Mike Franklin, Hector Garcia Molina, Dieter Gawlick, Jim Gray, Laura Haas, Alon Halevy, Joe Hellerstein, Yannis Ioannidis, Martin Kersten, Michael Pazzani, Mike Lesk, David Maier, Jeff Naughton, Hans Schek, Timos Sellis, Avi Silberschatz, Mike Stonebraker, Rick Snodgrass, Jeff Ullman, Gerhard Weikum, Jennifer Widom, and Stan Zdonik. The Lowell database research self-assessment. *Commun. ACM*, 48(5):111–118, 2005.

[3] Noga Alon, Tova Milo, Frank Neven, Dan Suciu, and Victor Vianu. Typechecking XML views of relational databases. *ACM Transactions on Computational Logic*, 4(3):315–354, 2003.

[4] Noga Alon, Tova Milo, Frank Neven, Dan Suciu, and Victor Vianu. XML with data values: typechecking revisited. *Journal of Computer and System Sciences*, 66(4):688–727, 2003.

[5] Henk P. Barendregt. *The Lambda Calculus: its Syntax and Semantics*. North-Holland, 1984.

[6] Scott Boag, Don Chamberlin, Mary F. Fernández, Daniela Florescu, Jonathan Robie, and Jérôme Siméon. *XQuery 1.0: An XML Query Language*. W3C Working Draft, February 2005.

[7] Don Box and Anders Hejlsberg. The LINQ Project: .NET Language Integrated Query. `http://msdn.microsoft.com/netframework/future/linq/`, March 2006.

[8] Peter Buneman, Susan B. Davidson, Mary F. Fernandez, and Dan Suciu. Adding structure to unstructured data. In Foto N. Afrati and Phokion Kolaitis, editors, *Database Theory—ICDT'97, 6th International Conference*, volume 1186, pages 336–350, Delphi, Greece, 1997. Springer.

[9] Peter Buneman, Shamim A. Naqvi, Val Tannen, and Limsoon Wong. Principles of programming with complex objects and collection types. *Theoretical Computer Science*, 149(1):3–48, 1995.

[10] Peter Buneman and Atsushi Ohori. Polymorphism and type inference in database programming. *ACM Transactions on Database Systems*, 21(1):30–76, 1996.

[11] C. Calcagno, E. Moggi, and T. Sheard. Closed types for a safe imperative MetaML. *Journal of Functional Programming*, 13(3):545–571, 2003.

[12] Luca Cardelli. Type systems. In *The Computer Science and Engineering Handbook*, pages 2208–2236. CRC Press, 1997.

[13] R. G. G. Cattell, Douglas K. Barry, Mark Berler, Jeff Eastman, David Jordan, Craig Russell, Olaf Schadow, Torsten Stanienda, , and Fernando Velez, editors. *The Object Data Standard: ODMG 3.0*. Morgan Kaufmann, 2000.

[14] W. Chen, M. Kifer, and D.S. Warren. HiLog: A foundation for higher-order logic programming. *Journal of Logic Programming*, 15(3):187–230, 1993.

[15] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *POPL '82: Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–212, New York, NY, USA, 1982. ACM Press.

[16] Denise Draper, Peter Fankhauser, Mary F. Fernández, Ashok Malhotra, Kristoffer Rose, Michael Rys, Jérôme Siméon, and Philip Wadler. *XQuery 1.0 and XPath 2.0 Formal Semantics*. W3C Working Draft, February 2005.

[17] Mary F. Fernández, Ashok Malhotra, Jonathan Marsh, Marton Nagy, and Norman Walsh. *XQuery 1.0 and XPath 2.0 Data Model*. W3C Working Draft, February 2005.

[18] Alain Frisch, Giuseppe Castagna, and Véronique Benzaken. ℂDuce: an XML-centric general-purpose language. In *Proceedings of the eighth ACM SIGPLAN International Conference on Functional Programming*, pages 51–63. ACM Press, 2003.

[19] D. Gawlick, D. Lenkov, A. Yalamanchi, et al. Applications for expression data in relational database systems. In *Proceedings 20th International Conference on Data Engineering*, pages 609–620. IEEE Computer Society, 2004.

[20] Roy Goldman and Jennifer Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In *VLDB'97 Proceedings of*

23rd International Conference on Very Large Data Bases, pages 436–445, 1997.

[21] M. Gyssens and L.V.S. Lakshmanan. A foundation for multi-dimensional databases. In *Proceedings 23rd International Conference on Very Large Data Bases*, pages 106–115. Morgan Kaufmann, 1997.

[22] Roger Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, December 1969.

[23] Haruo Hosoya and Benjamin C. Pierce. XDuce: A statically typed XML processing language. *ACM Transactions on Internet Technology (TOIT)*, 3(2):117–148, 2003.

[24] Haruo Hosoya, Jérôme Vouillon, and Benjamin C. Pierce. Regular expression types for XML. *ACM Transactions on Programming Languages and Systems*, 27(1):46–90, 2005.

[25] M. Jain, A. Mendhekar, and D. Van Gucht. A uniform data model for relational data and meta-data query processing. In *Advances in Data Management '95*, pages 146–165. Tata McGraw-Hill, 1995.

[26] Christoph Koch. On the complexity of nonrecursive xquery and functional query languages on complex values. *ACM Trans. Database Syst.*, 31(4):1215–1256, 2006.

[27] L.V.S. Lakshmanan, F. Sadri, and S.N. Subramanian. SchemaSQL: An extension to SQL for multidatabase interoperability. *ACM Transactions on Database Systems*, 26(4):476–519, 2001.

[28] Zi Lin, Bingsheng He, and Byron Choi. A quantitative summary of XML structures. In *ER 2006 - 25th International Conference on Conceptual Modeling*, volume 4215 of *Lecture Notes in Computer Science*, pages 228–240. Springer, 2006.

[29] Sebastian Maneth, Alexandru Berlea, Thomas Perst, and Helmut Seidl. XML type checking with macro tree transducers. In Chen Li, editor, *PODS*, pages 283–294. ACM, 2005.

[30] Sebastian Maneth, Thomas Perst, and Helmut Seidl. Exact XML type checking in polynomial time. In Schwentick and Suciu [41], pages 254–268.

[31] Wim Martens and Frank Neven. Frontiers of tractability for typechecking simple XML transformations. In *Proceedings of the Twenty-third ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 23–34. ACM Press, 2004.

[32] Wim Martens and Frank Neven. On the complexity of typechecking top-down XML transformations. *Theor. Comput. Sci.*, 336(1):153–180, 2005.

[33] Jim Melton. *Understanding SQL's Stored Procedures*. Morgan Kaufmann, San Mateo, CA, USA, 1998.

[34] Robin Milner. A theory of type polymorphism in programming. *J. Comput. Syst. Sci.*, 17(3):348–375, 1978.

[35] Tova Milo, Dan Suciu, and Victor Vianu. Typechecking for XML transformers. In *Proceedings of the Nineteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 11–22. ACM Press, 2000.

[36] F. Neven, J. Van den Bussche, D. Van Gucht, and G. Vossen. Typed query languages for databases containing queries. *Information Systems*, 24(7):569–595, 1999.

[37] Atsushi Ohori. A polymorphic record calculus and its compilation. *ACM Transactions on Programming Languages and Systems*, 17(6):844–895, 1995.

[38] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[39] Didier Rémy. Type inference for records in a natural extension of ML. In Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects Of Object-Oriented Programming. Types, Semantics and Language Design*. MIT Press, 1993.

[40] K. Ross. Relations with relation names as arguments: Algebra and calculus. In *Proceedings 11th ACM Symposium on Principles of Database Systems*, pages 346–353, 1992.

[41] Thomas Schwentick and Dan Suciu, editors. *Database Theory - ICDT 2007, 11th International Conference, Barcelona, Spain, January 10-12, 2007, Proceedings*, volume 4353 of *Lecture Notes in Computer Science*. Springer, 2007.

[42] H. Schwichtenberg. Definierbare funktionen in λ-kalkül mit typen. *Archiv für mathematische Logik und Grundlagenforschung*, 174:113–114, 1976.

[43] M. Stonebraker et al. QUEL as a data type. In B. Yormark, editor, *Proceedings of SIGMOD 84 Annual Meeting*, volume 14:2 of *SIGMOD Record*, pages 208–214. ACM Press, 1984.

[44] M. Stonebraker et al. Extending a database system with procedures. *ACM Transactions on Database Systems*, 12(3):350–376, 1987.

[45] Dan Suciu. Typechecking for semistructured data. In *Database Programming Languages, 8th International Workshop, DBPL 2001, Revised Papers*, volume 2397 of *Lecture Notes in Computer Science*, pages 1–20. Springer-Verlag, 2001.

[46] Martin Sulzmann. *A General Framework for Hindley/Milner Type Systems with Constraints*. PhD thesis, Yale University, 2000.

[47] Martin Sulzmann. A general type inference framework for Hindley/Milner style systems. In *Functional and Logic Programming: FLOPS 2001*, volume 2024 of *Lecture Notes in Computer Science*, pages 248–263. Springer-Verlag, 2001.

[48] W. Taha and T. Sheard. MetaML and multi-stage programming with explicit annotations. *Theoretical Computer Science*, 248(1–2):211–242, 2000.

[49] Wolfgang Thomas. *Languages, Automata, and Logic*, volume 3 of *Handbook of Formal Languages*, chapter 7, pages 389–456. Springer, 1997.

[50] Henry S. Thompson, David Beech, Murray Maloney, and Noah Mendelsohn. *XML Schema Part 1: Structures*. W3C Recommendation, May 2001.

[51] J. Van den Bussche, D. Van Gucht, and G. Vossen. Reflective programming in the relational algebra. *Journal of Computer and System Sciences*, 52(3):537–549, June 1996.

[52] J. Van den Bussche, S. Vansummeren, and G. Vossen. Meta-SQL: Towards practical meta-querying. In *Advances in Database Technology—EDBT 2004*,

volume 2992 of *Lecture Notes in Computer Science*, pages 823–825. Springer, 2004.

[53] J. Van den Bussche, S. Vansummeren, and G. Vossen. Towards practical meta-querying. *Information Systems*, 30(4):317–332, 2005.

[54] J. Van den Bussche and E. Waller. Polymorphic type inference for the relational algebra. *Journal of Computer and System Sciences*, 64:694–718, 2002.

[55] Jan Van den Bussche, Dirk Van Gucht, and Stijn Vansummeren. Well-definedness and semantic type-checking for the nested relational calculus. *Theoretical Computer Science*, 371(3):183–199, 2007.

[56] Jan Van den Bussche and Stijn Vansummeren. Polymorphic type inference for the named nested relational calculus. *ACM Transactions on Computational Logic*, To appear.

[57] Stijn Vansummeren. Deciding well-definedness of XQuery fragments. In *Proceedings of the twenty-fourth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 37–48, New York, NY, USA, 2005. ACM Press.

[58] Stijn Vansummeren. On deciding well-definedness for query languages on trees. Technical report, Hasselt University, 2005.

[59] Stijn Vansummeren. On the complexity of deciding typability in the relational algebra. *Acta Informatica*, 41(6):367–381, 2005.

[60] M. Vardi. The complexity of relational query languages. In *Proceedings 14th ACM Symposium on the Theory of Computing*, pages 137–146, 1982.

[61] Mitchell Wand. Type inference for record concatenation and multiple inheritance. *Information and Computation*, 93(1):1–15, 1991.

[62] Piotr Wieczorek. Complexity of typechecking XML views of relational databases. In Schwentick and Suciu [41], pages 239–253.

[63] Limsoon Wong. Kleisli, a functional query system. *Journal of Functional Programming*, 10(1):19–56, 2000.

[64] C.M. Wyss and E.L. Robertson. Relational languages for metadata integration. *ACM Transactions on Database Systems*, 30(2):624–660, 2005.

[65] XML syntax for XQuery 1.0 (XQueryX). W3C Working Draft 19 December 2003.