Polymorphic type inference for the relational algebra

Jan Van den Bussche
Limburgs Universitair Centrum
Belgium
jan.vandenbussche@luc.ac.be

Emmanuel Waller
LRI, Université Paris Sud
France
emmanuel.waller@lri.fr

Abstract

We give a polymorphic account of the relational algebra. We introduce a formalism of "type formulas" specifically tuned for relational algebra expressions, and present an algorithm that computes the "principal" type for a given expression. The principal type of an expression is a formula that specifies, in a clear and concise manner, all assignments of types (sets of attributes) to relation names, under which a given relational algebra expression is well-typed, as well as the output type that expression will have under each of these assignments. Topics discussed include complexity and polymorphic expressive power.

1 Introduction

The operators of the relational algebra (the basis of all relational query languages) are polymorphic. We can take the natural join of any two relations, regardless of their sets of attributes. We can take the union of any two relations over the same set of attributes. We can take the cartesian product of

any two relations having no attributes in common. We can perform a selection $\sigma_{A<B}$ on any relation having at least the attributes A and B. Similar typing conditions can be formulated for the other operators of the relational algebra. When combining operators into expressions, these typing conditions can become more involved. For example, for the expression

$$\sigma_{A<5}(r \bowtie s) \bowtie ((r \times u) - v)$$

to be well-typed, the attribute A must be an attribute of r or s (or both). But if it is an attribute of r, then it must also be one of v. Moreover, by the subexpression $(r \times u) - v$, the relation schemas of r and s must be disjoint, and their union must be the type of v.

A natural question thus arises: given a relational algebra expression e, under which database schemas is e well-typed? And what is the result relation schema of e under each of these assignments? This is nothing but the relational algebra version of the classical *type inference* problem. Type inference is an extensively studied topic in the theory of programming languages [1, 7, 9, 15, 5], and is used in industrial-strength functional programming languages such as SML/NJ [16].

Doing type inference for some language involves setting up two things. First, we need a system of *type rules* that allow to derive the output type of a program given types for its input parameters. Typically such an output type can only be derived for some of all possible assignments of types to input parameters; under these assignments the program is said to be *well-typed.* Second, we need a formalism of *type formulas.* A type formula defines a family of input type assignments, as well as an output type for each type assignment in the family. Every typable program should have a *principal* type formula, which defines all type assignments under which the program is well-typed, as well as the output type of the program under each of these assignments. The task then is to come up with a *type inference* algorithm that will compute the principal type for any given program.

In this paper, we do type inference for the relational algebra. The relational algebra is very different from the programming languages usually considered in type inference; two fundamental features of such languages, higher-order functions and data constructors (function symbols) are completely absent here. On the other hand, the set-based nature of relation types, and the particulars of the standard relational algebra operators when viewed polymorphically, present new challenges. As a consequence, our for-

malism of type formulas is drastically different from the formalisms used in the theory of programming languages.

Our main motivation for this work was foundational and theoretical; after all, query languages are specialized programming languages, so important ideas from programming languages should be applied and adapted to the query language context as much as possible. However, we also believe that type inference for database query languages is tied to the familiar 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, as long as these changes are to parts of the schema on which the query does not depend. To give a trivial example, the SQL query `select * from R where A<5` still works if we drop from `R` some column `B` different from `A`, but not if we drop column `A` itself. Turning this around, it is thus useful to infer, given a query, under exactly which schemas it works, so that the programmer sees to which schema changes the query is sensitive.

Some recent trends in database systems seem to add weight to the above motivation. *Stored procedures* [8] 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. Knowing the principal type of each stored procedure may be helpful in this regard. Models of *semi-structured* data [4, 3] loosen (or completely abandon) the assumption of a given fixed schema. Query languages for these models are essentially schema-independent. Nevertheless, as argued by Buneman et al. [2], querying is more effective if at least some form of schema is available, computed from the particular instance. Type inference can be helpful in telling for which schemas a given query is suitable.

Ohori, Buneman and Breazu-Tannen were probably the first to introduce type inference in the context of database programming languages, in their work on the language Machiavelli [11, 10]. Machiavelli features polymorphic field selection from nested records, as well as a polymorphic join operator. However, the inference of principal types for full-fledged relational algebra expressions was not taken up in that work. We should also mention the work of Stemple et al. [14], who investigated reflective implementations of the polymorphic relational algebra operators.

Other important related work is that on the extension of functional programming languages with polymorphic record types. Some of the most sophisticated proposals in that direction were made by Rémy [12, 13]. This

work adds record types to the type system of ML, featuring polymorphic field selection and record concatenation. While this system captures many realistic functional programs involving records, it cannot express the conditions on the types of relations implied by certain relational algebra expressions, such as the example we gave earlier. Notably constraints such as set disjointness (needed for the operator $\times$) or set equality (for the operator $\cup$), cannot be expressed in other systems. The reason is probably the additional concern of these systems for subtyping: a program applicable to records of a certain type should more generally be applicable to records having all the fields of that type and possibly more. This is clearly not true for relational algebra expressions.

If one is only interested in deciding whether a given relational algebra expression is *typable* (i.e., whether there exists at least one schema under which the expression is well-typed), we show that this problem is in the complexity class NP.

In a final section of this paper, we formally define the notion of *polymorphic query*. Using our type inference algorithm, we prove that various operators usually considered "derived," because they can be simulated using the standard relational algebra operators (e.g., semijoin), can *not* be simulated *in a polymorphic way*. Thus, our work also brings up new issues in the design of appropriate polymorphic query languages.

2 Preliminaries

2.1 Schemas, types, and expressions

Assume given sufficiently large supplies of *relation variables* and of *attribute names*. Relation variables will be denoted by lowercase letters from the end of the alphabet. Attribute names will be denoted by uppercase letters from the beginning of the alphabet.

A *schema* is a finite set $\mathcal{S}$ of relation variables. A *type* is a finite set τ of attribute names. Let $\mathcal{S}$ be a schema. A *type assignment on* $\mathcal{S}$ is a mapping $\mathcal{T}$ on $\mathcal{S}$, assigning to each $r \in \mathcal{S}$ a type $\mathcal{T}(r)$. So, we have split the usual notion of database schema, which specifies both the relation names and the associated sets of attributes, in two notions.

The expressions of the *relational algebra* are defined by the following

grammar:

$$\begin{array}{rcl} e & \rightarrow & r \\ & | & (e \cup e) \mid (e - e) \mid (e \bowtie e) \mid (e \times e) \\ & | & \sigma_{\theta(A_1,\ldots,A_n)}(e) \mid \pi_{A_1,\ldots,A_n}(e) \mid \rho_{A/B}(e) \mid \widehat{\pi}_A(e) \end{array}$$

Here e denotes an expression, r denotes a relation variable, and A, B, and A_i denote attribute names. The θ denotes a selection predicate.

The schema consisting of all relation variables occurring in expression e is denoted by $\mathit{Relvars}(e)$.

2.2 Well-typed expressions

Let $\mathcal{S}$ be a schema, e an expression with $\mathit{Relvars}(e) \subseteq \mathcal{S}$, $\mathcal{T}$ a type assignment on $\mathcal{S}$, and τ a type. The rules for when e *has type* τ *given* $\mathcal{T}$, denoted by $\mathcal{T} \vdash e : \tau$, are the following:

$$\frac{\mathcal{T}(r) = \tau}{\mathcal{T} \vdash r : \tau} \qquad \frac{\mathcal{T} \vdash e_1 : \tau \quad \mathcal{T} \vdash e_2 : \tau}{\mathcal{T} \vdash (e_1 \cup e_2) : \tau} \qquad \frac{\mathcal{T} \vdash e_1 : \tau \quad \mathcal{T} \vdash e_2 : \tau}{\mathcal{T} \vdash (e_1 - e_2) : \tau}$$

$$\frac{\mathcal{T} \vdash e_1 : \tau_1 \quad \mathcal{T} \vdash e_2 : \tau_2}{\mathcal{T} \vdash (e_1 \bowtie e_2) : \tau_1 \cup \tau_2} \qquad \frac{\mathcal{T} \vdash e_1 : \tau_1 \quad \mathcal{T} \vdash e_2 : \tau_2 \quad \tau_1 \cap \tau_2 = \varnothing}{\mathcal{T} \vdash (e_1 \times e_2) : \tau_1 \cup \tau_2}$$

$$\frac{\mathcal{T} \vdash e : \tau \quad A_1, \ldots, A_n \in \tau}{\mathcal{T} \vdash \sigma_{\theta(A_1,\ldots,A_n)}(e) : \tau} \qquad \frac{\mathcal{T} \vdash e : \tau \quad A_1, \ldots, A_n \in \tau}{\mathcal{T} \vdash \pi_{A_1,\ldots,A_n}(e) : \{A_1, \ldots, A_n\}}$$

$$\frac{\mathcal{T} \vdash e : \tau \quad A \in \tau \quad B \notin \tau}{\mathcal{T} \vdash \rho_{A/B}(e) : (\tau - \{A\}) \cup \{B\}} \qquad \frac{\mathcal{T} \vdash e : \tau \quad A \in \tau}{\mathcal{T} \vdash \widehat{\pi}_A(e) : \tau - \{A\}}$$

We have a first basic definition:

Definition 1 Let e be an expression and let $\mathcal{T}$ be a type assignment on $\mathit{Relvars}(e)$. If there exists a type τ such that $\mathcal{T} \vdash e : \tau$, we say that e is *well-typed under* $\mathcal{T}$.

Note that in this case τ is unique and can easily be derived from $\mathcal{T}$ by applying the rules in an order determined by the syntax of the expression e.

2.3 Semantics

We assume given a universe $\mathbf{U}$ of *data elements*.

Let τ be a type. A *tuple of type* τ is a mapping $\mathbf{t}$ on τ, assigning to each $A \in \tau$ a data element $\mathbf{t}(A) \in \mathbf{U}$. A *relation of type* τ is a finite set of tuples of type τ.

Let $\mathcal{S}$ be a schema, and let $\mathcal{T}$ be a type assignment on $\mathcal{S}$. A *database of type* $\mathcal{T}$ is a mapping $\mathbf{D}$ on $\mathcal{S}$, assigning to each $r \in \mathcal{S}$ a relation $\mathbf{D}(r)$ of type $\mathcal{T}(r)$.

The semantics of well-typed relational algebra expressions is the well-known one. If $\mathcal{T} \vdash e : \tau$, and $\mathbf{D}$ is a database of type $\mathcal{T}$, then the *result of evaluating* e *on* $\mathbf{D}$ is a relation of type τ defined in the well-known manner. The only operator worth mentioning is perhaps the not so usual $\widehat{\pi}_A$, which projects *out* the attribute A, leaving all others intact.

At this point a remark is in order concerning the non-redundancy of the set of relational operators we consider. We have included both the natural join $\bowtie$ and the cartesian product $\times$, and also both the standard projection $\pi_{A_1,\ldots,A_n}$ and the "complementary" projection $\widehat{\pi}_A$. It is well known that if the type assignment is fixed and known, $\bowtie$ can be simulated using $\times$ (plus selection and renaming), and conversely, $\times$ can be simulated using $\bowtie$ (plus renaming). Also, π can be simulated by a series of $\widehat{\pi}$'s, and $\widehat{\pi}$ can be simulated by π. To illustrate the latter, if we fix the type of r to $\{A, B, C\}$, then $\pi_A(r)$ is equivalent to $\widehat{\pi}_B\widehat{\pi}_C(r)$, and $\widehat{\pi}_A(r)$ is equivalent to $\pi_{B,C}(r)$. However, these simulations are not "polymorphic," in the sense that they depend on the particular type assignment.

As a matter of fact, we will see in Proposition 2 that *polymorphic* simulations of $\bowtie$ using $\times$, or vice versa, and of π using $\widehat{\pi}$, or vice versa, do not exist. Hence, from a polymorphic point of view, our chosen set of relational algebra operators is *non-redundant.*

3 Typable expressions

The central notion of this paper is defined as follows:

Definition 2 Expression e is called *typable* if there exists a type assignment $\mathcal{T}$ on $Relvars(e)$ such that e is well-typed under $\mathcal{T}$.

A very simple example of an expression that is not typable is $\sigma_{A=B}(\pi_{B,C}(r))$.

Is typability a decidable property? This question is easily answered by the following lemma. We use the following notation. If $\mathcal{T}$ is a type assignment and $\mathcal{A}$ is a set of attribute names, then we denote by $\mathcal{T}|_{\mathcal{A}}$ the type assignment defined by $\mathcal{T}|_{\mathcal{A}}(r) := \mathcal{T}(r) \cap \mathcal{A}$. If e is an expression then we denote the set of all attribute names that explicitly occur in e by $\mathit{Specattrs}(e)$.

Lemma 1 *If $\mathcal{T} \vdash e : \tau$ and $\mathcal{A} \supseteq \mathit{Specattrs}(e)$, then $\mathcal{T}|_{\mathcal{A}} \vdash e : \tau \cap \mathcal{A}$.*

The proof is straightforward. As a consequence, in order to decide whether there exists a type assignment under which e is well-typed, it suffices to consider type assignments $\mathcal{T}$ with the property that $\mathcal{T}(r) \subseteq \mathit{Specattrs}(e)$ for every r. It follows immediately that typability is in NP. Whether or not it is in P, or is NP-complete, remains open.

Of course, we are not satisfied simply by knowing whether or not a given expression is typable. What we really want is a clear, concise picture of exactly under which type assignments it is well-typed, as well as of what type the expression will have under each of these type assignments. (Note that there will in general be infinitely many such type assignments.)

In the following, we will define the formalism of *type formulas*, which is specifically tuned towards this task.

4 Examples of type formulas

Consider the expression

$$e = \sigma_{B=C}((\rho_{A/B}(r) \cup s) \bowtie u).$$

This expression is well-typed under exactly those type assignments $\mathcal{T}$ satisfying the following two conditions:

1. $\mathcal{T}(s) = (\mathcal{T}(r) - \{A\}) \cup \{B\}$;
2. C must belong to at least one of $\mathcal{T}(u)$, $\mathcal{T}(r)$, or $\mathcal{T}(s)$.

Given such a $\mathcal{T}$, the type of e then will equal $\mathcal{T}(s) \cup \mathcal{T}(u)$.

All the above information is expressed by the following type formula for e:

$$\begin{array}{lcl} r : a_1a_2 & & \\ s : a_1a_2 & \mapsto & e : a_1a_2a_3 \\ u : a_2a_3 & & \\ A : r \wedge \neg s & & A : u \\ B : s \wedge \neg r & & B : \mathbf{true} \\ C : (r \leftrightarrow s) \wedge (r \vee s \vee u) & & C : \mathbf{true} \end{array}$$

This type formula will the output of our type inference algorithm. It can be intuitively read as follows. *Expression e is well-typed under precisely all type assignments that can be produced by the following procedure:*

1. *Instantiate a_1, a_2 and a_3 by any three types, on condition that they are pairwise disjoint, and do not contain A, nor B, nor C.*

2. Preliminarily *assign type $a_1 \cup a_2$ to r; $a_1 \cup a_2$ to s; and $a_2 \cup a_3$ to u.*

3. *In this preliminary type assignment, A must be added to the type of r, but must not be added to that of s; whether it is added to the type of u is a free choice.*

4. *Similarly, B must be added to the type of s, not to that of r, and freely to that of u.*

5. *Finally, C must be added at least to one of the types of r, s, and u, but if we add it to r we must also add it to s and vice versa.*

The type of e under a type assignment thus produced equals $a_1 \cup a_2 \cup a_3$, to which we must add B and C, and to which we also add A on condition that it belongs to the type of u.

The symbols a_1, a_2 and a_3 are called *type variables*. The attributes A, B and C, which are explicitly mentioned by the expression, are called the *special attributes* of the expression. The *declaration* of each relation variable as a string of type variables (where concatenation denotes union) provides the *polymorphic basis* of the type assignments under which the expression is well-typed. An *attribute constraint* for each special attribute then specifies (by a Boolean formula) the allowed extensions of the polymorphic basis types with that attribute. The declarations and constraints together form the *type context*; this is the left-hand side of the type formula. On the right-hand side we find the polymorphic basis of the output type, and again for each special

attribute, an *output condition* which specifies (by a Boolean formula) under which condition that attribute has to be added to the output type.

Let us see two more examples. The type formula for the expression

$$e = \pi_A(r) - \pi_A((\pi_A(r) \times s) - r),$$

which the reader will recognize as the textbook expression for the division operator, is:

$$\begin{array}{ll} r : a & \\ s : a & \mapsto \quad e : \varnothing \\ A : r \wedge \neg s & \qquad\quad A : \textbf{true} \end{array}$$

So r and s must have the same type except that r has an additional A (which s has not). The output type is always $\{A\}$.

The type formula for the expression discussed in the Introduction,

$$e = \sigma_{A<5}(r \bowtie s) \bowtie ((r \times u) - v),$$

is:

$$\begin{array}{ll} v : a_1a_2a_3a_4 & \\ r : a_1a_3 & \\ u : a_2a_4 & \mapsto \quad e : a_1a_2a_3a_4a_5 \\ s : a_3a_4a_5 & \\ A : (r \vee s) \wedge (v \leftrightarrow (r \vee u)) \wedge \neg(r \wedge u) & \qquad\quad A : \textbf{true} \end{array}$$

The declarations specify exactly, in a manner similar to Venn diagrams, the conditions required on the types of the relation variables for the expression to be well-typed.

5 Type formulas and type inference — Formal definitions

Before we can describe our type inference algorithm, we need precise definitions of the underlying formalism. In what follows, we assume given a sufficiently large supply of *type variables*.

5.1 Type contexts

A *type context* is a structure consisting of the following components:

1. A finite set *Relvars* of relation variables.
2. A finite set *Typevars* of type variables.
3. A mapping *decl* from *Relvars* to $2^{Typevars}$, called the *declaration mapping*.
4. A finite set *Specattrs* of attribute names (called the *special attributes*).
5. A mapping *constraint* on *Specattrs*, assigning to each special attribute a Boolean formula over *Relvars*.

We will usually denote a type context by the letter Γ and, when necessary to avoid ambiguities, will write $Relvars(\Gamma)$, $Typevars(\Gamma)$, etc.

5.2 Semantics of type contexts

Fix a type context Γ. The "models" of Γ will be type assignments on $Relvars(\Gamma)$. We go from type contexts to type assignments via the notion of instantiation. An *instantiation of* Γ is a mapping $\mathcal{I}$ on $Typevars \cup Specattrs$, such that

1. $\mathcal{I}$ assigns to each type variable a type, such that
   - for different type variables a_1 and a_2, $\mathcal{I}(a_1)$ and $\mathcal{I}(a_2)$ are disjoint; and
   - for each type variable a and special attribute A; $A \notin \mathcal{I}(a)$.
2. $\mathcal{I}$ assigns to each special attribute a subset of *Relvars*, such that for each special attribute A, $\mathcal{I}(A) \models constraint(A)$. (Since $constraint(A)$ is a Boolean formula over *Relvars*, and $\mathcal{I}(A)$ is a subset of *Relvars*, the meaning of $\mathcal{I}(A) \models constraint(A)$ is the standard meaning from propositional logic.)

If some of the Boolean formulas in Γ are unsatisfiable, we call also Γ *unsatisfiable*. In this case, Γ has no instantiations.

From a type context Γ and an instantiation $\mathcal{I}$ of Γ, we can uniquely determine a type assignment $\mathcal{T}$ on *Relvars*, defined on each relation variable r as follows:

$$\mathcal{T}(r) := \bigcup \{\mathcal{I}(a) \mid a \in decl(r)\} \cup \{A \in Specattrs \mid r \in \mathcal{I}(A)\}.$$

We call this type assignment $\mathcal{T}$ the *image of* Γ *under* $\mathcal{I}$, and conveniently denote it by $\mathcal{I}(\Gamma)$.

5.3 Type formulas

A *type formula* now is a quadruple $(\Gamma, e, \mathit{Outvars}, \mathit{outatt})$, where

1. Γ is a type context;
2. e is a relational algebra expression with $\mathit{Relvars}(e) = \mathit{Relvars}(\Gamma)$, and such that $\mathit{Specattrs}(\Gamma)$ contains all the attribute names that are explicitly mentioned in e.
3. *Outvars* is a subset of $\mathit{Typevars}(\Gamma)$; and
4. *outatt* is a mapping on $\mathit{Specattrs}(\Gamma)$, assigning to each special attribute a Boolean formula over $\mathit{Relvars}(\Gamma)$.

The way we write down concrete instances of type formulas has already been illustrated in Section 4.

5.4 Semantics of type formulas

From a type formula $(\Gamma, e, \mathit{Outvars}, \mathit{outatt})$ and an instantiation $\mathcal{I}$ of Γ, we can uniquely determine the following type:

$$\{\mathcal{I}(a) \mid a \in \mathit{Outvars}\} \cup \{A \in \mathit{Specattrs} \mid \mathcal{I}(A) \models \mathit{outatt}(A)\}.$$

We call this type the *output type of the type formula under* $\mathcal{I}$.

We are now ready to define the following fundamental property of type formulas:

Definition 3 A type formula $(\Gamma, e, \mathit{Outvars}, \mathit{outatt})$ is called *principal for* e if for every type assignment $\mathcal{T}$ on $\mathit{Relvars}(e)$ and every type τ, $\mathcal{T} \vdash e : \tau$ if and only if there is an instantiation $\mathcal{I}$ of Γ such that $\mathcal{T}$ is the image of Γ under $\mathcal{I}$, and such that τ is the output type of the type formula under $\mathcal{I}$.

The main result of this paper can now succinctly stated as follows:

Theorem 1 (Type inference) *For every relational algebra expression e, there exists a principal type formula for e, which can be effectively computed from e.*

Note that if e is untypable, any unsatisfiable type formula (type formula with an unsatisfiable type context) is principal for e.

We will substantiate our main theorem in the following sections.

6 Solving systems of set equations

Type inference algorithms for programming languages typically work by structural induction on program expressions, enforcing the typing rules "in reverse," and using some form of unification to combine type formulas of subexpressions. In our case, relation types are sets, so we need a replacement for classical unification on terms. This role will be played by the following algorithm for solving systems of set equations.

Fix some universe $\mathcal{U}$. In principle $\mathcal{U}$ can be any set, but in our intended application $\mathcal{U}$ is the universe of attribute names. Assume further given a sufficiently large supply of *variables*. In our intended application, this role will be played by type variables.

An *equation* is an expression of the form $lhs = rhs$, where both lhs and rhs are sets of variables.[1] A *system of equations* consists of two disjoint sets L and R of variables, and a set of equations, such that every variable occurring at the left-hand side (right-hand side) of some equation is in L (in R).

A *substitution* on a set S of variables is a mapping from S to the subsets of $\mathcal{U}$. A substitution is called *proper* if different variables are assigned disjoint sets. A *valuation* of a system Σ consists of a proper substitution on L and a proper subsitution on R. A valuation (f_L, f_R) is a *solution of* Σ if for every equation

$$a_1 \dots a_m = b_1 \dots b_n$$

in Σ, we have

$$f_L(a_1) \cup \dots \cup f_L(a_m) = f_R(b_1) \cup \dots \cup f_R(b_n).$$

A *symbolic valuation* of Σ consists of a new set V of variables and a mapping g from $L \cup R$ to the subsets of V. Take some proper substitution h on V. Now define the following substitution h_L on L: for any $a \in L$,

$$h_L(a) := \bigcup \{h(c) \mid c \in g(a)\}.$$

In a completely analogous way we also define the substitution h_R on R. We call a symbolic valuation a *symbolic solution of* Σ if for every proper substitution h on V, the pair (h_L, h_R) is a solution of Σ, and conversely, every solution of Σ can be written in this way. So, a symbolic solution is a finite representation of the set of all solutions.

[1] A note on notation: we will write a set $\{a_1, \dots, a_n\}$ as $a_1 \dots a_n$.

As a trivial example, consider the trivial system of equations where $L = \{a\}$, $R = \{b\}$, and without any equations. Any valuation is also a solution. A symbolic solution is given by $V = \{c_1, c_2, c_3\}$ and

$$g(a) = c_1 c_2 \qquad \text{and} \qquad g(b) = c_2 c_3.$$

Indeed, note that we always work with proper substitutions, so c_1, c_2 and c_3 stand for pairwise disjoint sets. In particular, c_1 stands for $a - b$, c_2 stands for $a \cap b$, and c_3 stands for $b - a$.

Theorem 2 *Every system of equations Σ has a symbolic solution, which can be computed from Σ in polynomial time.*

Proof. Let

$$V := \{\bar{a} \mid a \in L\} \cup \{\bar{b} \mid b \in R\} \cup \{(\bar{a}, \bar{b}) \mid (a, b) \in (L \times R)\},$$

and define the following symbolic valuation g with V as its set of variables: for each $a \in L$,

$$g(a) := \{(\bar{a}, \bar{b}) \mid b \in R\} \cup \{\bar{a}\}$$

and for each $b \in R$,

$$g(b) := \{(\bar{a}, \bar{b}) \mid a \in L\} \cup \{\bar{b}\}.$$

Then define the subset $V_0 \subseteq V$ as follows. An element $c \in V$ is in V_0 if there is an equation

$$a_1 \ldots a_m = b_1 \ldots b_n$$

in Σ such that c belongs to one of the following two sets but not to the other:

$$\bigcup_{i=1}^{m} g(a_i) \qquad \text{and} \qquad \bigcup_{j=1}^{n} g(b_j).$$

Now consider the symbolic valuation g' with $V' := V - V_0$ as its set of variables, defined by $g'(x) := g(x) - V_0$. This g' can easily be constructed in polynomial time. We next show that g' is indeed a symbolic solution of Σ.

Let h be a proper substitution on V', and let $a_1 \ldots a_m = b_1 \ldots b_n$ be an equation. By definition of g', for every $i \in \{1, \ldots, m\}$ and every $c \in g'(a_i)$, there is a $j \in \{1, \ldots, n\}$ such that $c \in g'(b_j)$, and vice versa, for every

$j \in \{1, \ldots, n\}$ and every $c \in g'(b_j)$, there is an $i \in \{1, \ldots, n\}$ such that $c \in g'(a_i)$. Hence,

$$\bigcup_{i=1}^{m} \underbrace{\bigcup \{h(c) \mid c \in g'(a_i)\}}_{h_L(a_i)} = \bigcup_{j=1}^{m} \underbrace{\bigcup \{h(c) \mid c \in g'(b_j)\}}_{h_R(b_j)}$$

and thus (h_L, h_R) is a solution of Σ.

Conversely, let (f_L, f_R) be a solution of Σ. Then define the following proper valuation h on V: for $a \in L$,

$$h(\bar{a}) := f_L(a) - \bigcup f_R(R);$$

for $b \in R$,

$$h(\bar{b}) := f_R(b) - \bigcup f_L(L);$$

and for $(a, b) \in (L \times R)$,

$$h(\bar{a}, \bar{b}) := f_L(a) \cap f_R(b).$$

Clearly, for each $a \in L$,

$$f_L(a) = \bigcup \{h(\bar{a}, \bar{b}) \mid b \in R\} \cup h(\bar{a}),$$

and for each $b \in R$,

$$f_R(b) = \bigcup \{h(\bar{a}, \bar{b}) \mid a \in L\} \cup h(\bar{b}).$$

Put differently,

$$f_L(a) = \bigcup \{h(c) \mid c \in g(a)\}$$

for each a, and

$$f_R(b) = \bigcup \{h(c) \mid c \in g(b)\}$$

for each b. Since we want to show that g' is a symbolic solution, we would like to show the last two equalities with g' instead of g. Since $g'(x) = g(x) - V_0$, it suffices to show that $h(c)$ is empty for each $c \in V_0$,

To see that these sets are indeed empty, we consider the three possibilities for an element of V to be in V_0. If $\bar{a} \in V_0$ with $a \in L$, this means that there is some equation

$$a_1 \ldots a_m = b_1 \ldots b_n$$

where a is one of the $a_1, \ldots, a_m$. Since (f_L, f_R) is a solution,

$$f_L(a) \subseteq \bigcup_{j=1}^{m} f_R(b),$$

so in particular, since $h(\bar{a}) \subseteq f_L(a)$,

$$h(\bar{a}) \subseteq \bigcup_{j=1}^{m} f_R(b).$$

However, by definition of h, $h(\bar{a})$ is disjoint from each $f_R(b)$. Hence, $h(\bar{a})$ must be empty.

Analogously we see that if $\bar{b} \in V_0$ with $b \in R$, then $h(\bar{b})$ is empty.

So finally, assume $(\bar{a}, \bar{b}) \in V_0$ with $(a, b) \in (L \times R)$. This means that there is either an equation of the form

$$\ldots a \ldots = \ldots$$

with b not occurring in the right-hand side, or of the form

$$\ldots = \ldots b \ldots$$

with a not occurring in the left-hand side. Let us focus on the first possibility (the second is analogous) and write the equation in more detail as

$$\ldots a \ldots = b_1 \ldots b_m.$$

Since (f_L, f_R) is a solution, $f_L(a)$, and in particular $f_L(a) \cap f_R(b)$, is contained in $\bigcup_{j=1}^{m} f_R(b_j)$. However, since b is not among $b_1, \ldots, b_m$, and each $f_R(b_j)$ is disjoint from $f_R(b)$, this can only be if $f_L(a) \cap f_R(b)$, which is the same as $h(\bar{a}, \bar{b})$, is empty. ■

Let us see a worked-out example of this solution method. Consider Σ with $L = \{a_1, a_2, a_3\}$, $R = \{b_1, b_2, b_3\}$, and the equations

$$a_1 = b_1 \qquad \text{and} \qquad a_2 = b_1 b_2.$$

From the first equation we deduce that

$$\bar{a}_1, \ (\bar{a}_1, \bar{b}_2), \ (\bar{a}_1, \bar{b}_3)$$

as well as

$$\bar{b}_1,\ (\bar{a}_2, \bar{b}_1),\ (\bar{a}_3, \bar{b}_1)$$

are in V_0. From the second equation we deduce that

$$\bar{a}_2,\ (\bar{a}_2, \bar{b}_3)$$

as well as

$$(\bar{a}_1, \bar{b}_1),\ \bar{b}_2,\ (\bar{a}_3, \bar{b}_2)$$

are also in V_0. So

$$V - V_0 = \{\bar{a}_3, \bar{b}_3, (\bar{a}_2, \bar{b}_2), (\bar{a}_3, \bar{b}_3)\},$$

and the symbolic solution g' is given by

$$\begin{array}{ll} g'(a_1) = \varnothing & g'(b_1) = \varnothing \\ g'(a_2) = (\bar{a}_2, \bar{b}_2) & g'(b_2) = (\bar{a}_2, \bar{b}_2) \\ g'(a_3) = \bar{a}_3,\ (\bar{a}_3, \bar{b}_3) & g'(b_3) = \bar{b}_3,\ (\bar{a}_3, \bar{b}_3). \end{array}$$

If we rename the variables for added clarity, we obtain the symbolic solution

$$\begin{array}{ll} a_1 = \varnothing & b_1 = \varnothing \\ a_2 = c_1 & b_2 = c_1 \\ a_3 = c_2 c_3 & b_3 = c_3 c_4 \end{array}$$

which can be interpreted as specifying that the only solutions to Σ are those where we assign the same set to a_2 and b_2, which is disjoint from the sets assigned to a_3 and b_3 (the latter two sets need not be disjoint), and where a_1 and b_1 are empty.

7 Principal type inference algorithm

We are now ready to describe our algorithm. A computer implementation is available from the authors [17].

7.1 Two subroutines

7.1.1 Extending a type formula with extra special attributes

The following construction will be used as a subroutine in our algorithm. Let $(\Gamma, e, \mathit{Outvars}, \mathit{outatt})$ be a type formula, and let A be an attribute name not in $\mathit{Specattrs}$. By *extending this type formula with* A, we mean the following:

1. add A to *Specattrs*;

2. define *constraint*(A) as

$$(\bigvee_{r} r) \rightarrow \bigvee_{a \in Typevars} (\bigwedge_{a \in decl(r)} r \wedge \bigwedge_{a \notin decl(r)} \neg r);$$

3. define *outatt*(A) as

$$\bigvee \{r \mid decl(r) \subseteq Outvars\}.$$

7.1.2 Conjugating two type contexts.

This is another subroutine that will be used. Two type contexts Γ_1 and Γ_2 are called *compatible* if *(i)* $Typevars_1 = Typevars_2$; *(ii)* $decl_1$ and $decl_2$ agree on $Relvars_1 \cap Relvars_2$; and *(iii)* $Specattrs_1 = Specattrs_2$. By the *conjunction* of two compatible type contexts Γ_1 and Γ_2, we mean the type context defined as follows:

1. $Relvars := Relvars_1 \cup Relvars_2$.

2. $Typevars := Typevars_1$ $(= Typevars_2)$.

3. $decl := decl_1 \cup decl_2$.

4. $Specattrs := Specattrs_1$ $(= Specattrs_2)$.

5. for each $A \in Specattrs$,

$$constraint(A) := constraint_1(A) \wedge constraint_2(A).$$

7.2 The algorithm

7.2.1 Base case

Our algorithm proceeds by induction on the structure of the expression. The base case, where e is a relation variable r, is trivial:

$$r : a \mapsto r : a.$$

7.2.2 Union

Let $e = (e_1 \cup e_2)$. By induction, for $i = 1, 2$, we have principal type formulas $(\Gamma_i, e_i, \mathit{Outvars}_i, \mathit{outatt}_i)$. We may assume that $\mathit{Typevars}_1$ and $\mathit{Typevars}_2$ are disjoint. We perform the following steps:

1. For each A in $\mathit{Specattrs}_1$ not in $\mathit{Specattrs}_2$, extend the type formula for e_2 by A. Conversely, for each A in $\mathit{Specattrs}_2$ not in $\mathit{Specattrs}_1$, extend the type formula for e_1 by A. We now have $\mathit{Specattrs}_1 = \mathit{Specattrs}_2$, which we denote by $\mathit{Specattrs}$.

2. Now consider the system of set equations Σ with $L = \mathit{Typevars}_1$, $R = \mathit{Typevars}_2$, and the set of equations

   $$\{\mathit{decl}_1(r) = \mathit{decl}_2(r) \mid r \in \mathit{Relvars}_1 \cap \mathit{Relvars}_2\} \cup \{\mathit{Outvars}_1 = \mathit{Outvars}_2\}.$$

   Find a symbolic solution to this system, and apply it to the two type formulas. Denote the result of applying the solution to $\mathit{Outvars}_1$ by $\mathit{Outvars}$; by the equation $\mathit{Outvars}_1 = \mathit{Outvars}_2$, this is the same as the result of applying the solution to $\mathit{Outvars}_2$.

3. The two type contexts Γ_1 and Γ_2 have now become compatible; in particular, they have the same set of type variables, which we denote by $\mathit{Typevars}$. Take their conjunction Γ. The resulting set of relation variables is denoted by $\mathit{Relvars}$. The resulting constraint mapping is denoted by $\mathit{constraint}'$.

4. For each A in $\mathit{Specattrs}$, define $\mathit{constraint}(A)$ as

   $$\mathit{constraint}'(A) \wedge (\mathit{outatt}_1(A) \leftrightarrow \mathit{outatt}_2(A)),$$

   and define $\mathit{outatt}(A)$ as $\mathit{outatt}_1(A)$.

The result is a principal type formula $(\Gamma, e, \mathit{Outvars}, \mathit{outatt})$ for e.

7.2.3 Difference

The case $e = (e_1 - e_2)$ is treated in exactly the same way as the case $e = (e_1 \cup e_2)$.

7.2.4 Natural join

The case $e = (e_1 \bowtie e_2)$ is treated as the case $e = (e_1 \cup e_2)$, except for the following important differences in two of the steps:

2. We omit the equation $\mathit{Outvars}_1 = \mathit{Outvars}_2$ from the system of equations. We now define *Outvars* as the union of the results of applying the symbolic solution to $\mathit{Outvars}_1$ and $\mathit{Outvars}_2$.

4. For each A in *Specattrs*, $\mathit{constraint}(A)$ is now the same as $\mathit{constraint}'(A)$, and $\mathit{outatt}(A)$ is now defined as

$$\mathit{outatt}_1(A) \vee \mathit{outatt}_2(A).$$

7.2.5 Cartesian product

The case $e = (e_1 \times e_2)$ is treated as the case $e = (e_1 \bowtie e_2)$, except for the following two differences, again in steps 2 and 4:

2. In the computation of the symbolic solution, we put every pair $(\bar{a}, \bar{b})$ with $a \in \mathit{Outvars}_1$ and $b \in \mathit{Outvars}_2$ by default in V_0 (cf. the solution method described in the proof of Theorem 2). This will guarantee that the results of applying the solution to $\mathit{Outvars}_1$ and $\mathit{Outvars}_2$ will be disjoint.

4. For each A in *Specattrs*, define $\mathit{constraint}(A)$ as

$$\mathit{constraint}'(A) \wedge \neg(\mathit{outatt}_1(A) \wedge \mathit{outatt}_2(A)).$$

7.2.6 Selection

Let $e = \sigma_{\theta(A_1,\ldots,A_n)}(e')$.

1. Initialize the desired type formula

$$(\Gamma, e, \mathit{Outvars}, \mathit{outatt})$$

   to the principal type formula $(\Gamma', e', \mathit{Outvars}', \mathit{outatt}')$ for e' (which we already have by induction).

2. For $i = 1, \ldots, n$, if A_i is not yet in *Specattrs*, extend the type formula with A_i.

3. for $i = 1, \ldots, n$, replace $\mathit{constraint}(A_i)$ by

$$\mathit{constraint}(A_i) \wedge \mathit{outatt}(A_i).$$

4. For $i = 1, \ldots, n$, put $\mathit{outatt}(A_i) := \mathbf{true}$.

7.2.7 Projection

For the case $e = \pi_{A_1,\ldots,A_n}(e')$ we do the same as for the case $e = \sigma_{\theta(A_1,\ldots,A_n)}(e')$. In addition, we set

- $\mathit{outatt}(A) := \mathbf{false}$ for each A in

$$\mathit{Specattrs} - \{A_1, \ldots, A_n\},$$

  and

- $\mathit{Outvars} := \varnothing$.

7.2.8 Renaming

The case $e = \rho_{A/B}(e')$ is treated similarly to the case $e = \sigma_{\theta(A,B)}(e')$, except that we treat B differently from A in step 3, as follows:

3. Replace $\mathit{constraint}(B)$ by

$$\mathit{constraint}(B) \wedge \neg \mathit{outatt}(B).$$

Furthermore, step 4 is changed as follows:

4. Put $\mathit{outatt}(A) := \mathbf{false}$, and $\mathit{outatt}(B) := \mathbf{true}$.

7.2.9 Projecting out

Finally, the case $e = \widehat{\pi}_A(e')$ is treated similarly to $e = \sigma_{\theta(A)}(e')$, with the exception that we set $\mathit{outatt}(A) := \mathbf{false}$ instead of $\mathbf{true}$.

7.3 Example

We illustrate the working of our algorithm on the expression

$$e = \sigma_{B=C}\,(\underbrace{(\underbrace{\underbrace{\rho_{A/B}(r)}_{e_1} \cup s}_{e_2}) \bowtie u}_{e_3})\,.$$

We will encounter only rather trivial systems of equations in doing this example; the reader is invited to try the example expression discussed in the Introduction for more interesting systems of equations.

To find the type formula for e_1, we start from the trival type formula $r : a \mapsto r : a$ for r. Extending this type formula with A and B yields

$$\begin{array}{lcl} r : a & \mapsto & r : a \\ A : r \to r & & A : r \\ B : r \to r & & B : r. \end{array}$$

Then we change the constraint $r \to r$ (or simply **true**) for A by $\mathbf{true} \wedge r$, or simply r, and we change the constraint for B by $\mathbf{true} \wedge \neg r$, or $\neg r$. Finally, we set $outatt(A)$ to **false** and $outatt(B)$ to **true**, yielding:

$$\begin{array}{lcl} r : a & \mapsto & e_1 : a \\ A : r & & A : \mathbf{false} \\ B : \neg r & & B : \mathbf{true}. \end{array}$$

To find the type formula for e_2, we start from that for e_1 and the trivial formula for s, which we extend with A and B as

$$\begin{array}{lcl} s : b & \mapsto & s : b \\ A : \mathbf{true} & & A : s \\ B : \mathbf{true} & & B : s. \end{array}$$

We now consider the rather trivial system of set equations with $L = \{a\}$, $R = \{b\}$, and the single equation $a = b$. The symbolic solution is obviously $a = c, b = c$. Applying this solution to the two type formulas simply changes both a and b into c. Conjugating the two type contexts yields the constraint $r \wedge \mathbf{true}$ for A, which can be simplified to r, and the constraint $\neg r \wedge \mathbf{true}$ for B, which can be simplified to $\neg r$. Then we add the conjunct $\mathbf{false} \leftrightarrow s$

to the constraint for A, yielding $r \wedge \neg s$, and we add the conjunct $\mathbf{true} \leftrightarrow s$ for B, yielding $\neg r \wedge s$. Finally, $outatt(A)$ is set to **false**, and $outatt(B)$ to **true**, yielding:

$$\begin{array}{lcl} r : c & & \\ s : c & \mapsto & e_2 : c \\ A : r \wedge \neg s & & A : \mathbf{false} \\ B : s \wedge \neg r & & B : \mathbf{true}. \end{array}$$

To find the type formula for e_3, we start from the one for e_2 and the trivial formula for u, which we extend with A and B as

$$\begin{array}{lcl} u : d & \mapsto & u : d \\ A : \mathbf{true} & & A : u \\ B : \mathbf{true} & & B : u. \end{array}$$

We now get the even more trivial system of set equations with $L = \{c\}$, $R = \{d\}$, and no equations, which has as symbolic solution $c = c_1c_2, d = c_2c_3$. We set *Outvars* to $c_1c_2c_3$. Conjugating the two type contexts (after having filled in the solution) yields nothing surprising. Finally we set $outatt(A)$ to $\mathbf{false} \vee u$, which simplifies to u, and set $outatt(B)$ to $\mathbf{true} \vee u$, or simply **true**, yielding:

$$\begin{array}{lcl} r : c_1c_2 & & \\ s : c_1c_2 & \mapsto & e_3 : c_1c_2c_3 \\ u : c_2c_3 & & \\ A : r \wedge \neg s & & A : u \\ B : s \wedge \neg r & & B : \mathbf{true}. \end{array}$$

Finally, to find the type formula for e itself, we first extend the one for e_3 with C:

$$\begin{array}{lcl} r : c_1c_2 & & \\ s : c_1c_2 & \mapsto & e_3 : c_1c_2c_3 \\ u : c_2c_3 & & \\ A : r \wedge \neg s & & A : u \\ B : s \wedge \neg r & & B : \mathbf{true} \\ C : \varphi & & C : r \vee s \vee u. \end{array}$$

Here, φ is the formula

$$(r \vee s \vee u) \rightarrow ((r \wedge s \wedge \neg u) \vee (r \wedge s \wedge u) \vee (\neg r \wedge \neg s \wedge u)),$$

or simply $r \leftrightarrow s$. Then we add the conjunct **true** to the constraint for B (which has no effect), and the conjunct $(r \vee s \vee u)$ to the constraint for

C. Finally, we set $outatt(B) = outatt(C) = \mathbf{true}$, yielding indeed the type formula we gave for e in Section 4 (modulo renaming of type variables).

7.4 Correctness proof

Extension of a type formula with extra special attributes (Section 7.1.1) is a heavily used subroutine in our type inference algorithm, and one might even go as far as saying that it is the only part of the algorithm whose correctness is not self-evident. Hence, the following lemma is of crucial importance:

Lemma 2 *The extension of any type formula, generated by our algorithm, with an extra special attribute, always produces an equivalent type formula.*

Here, equivalence naturally means the following. Consider two type formulas Φ_1 and Φ_2 whose type contexts Γ_1 and Γ_2 have the same *Relvars*, and let $\mathcal{I}_1$ ($\mathcal{I}_2$) be an instantiation of Γ_1 (Γ_2). We say that $\mathcal{I}_1$ *and* $\mathcal{I}_2$ *are equivalent with respect to* Φ_1 *and* Φ_2 if $\mathcal{I}_1(\Gamma_1) = \mathcal{I}_2(\Gamma_2)$, and the output type of Φ_1 under $\mathcal{I}_1$ equals the output type of Φ_2 under $\mathcal{I}_2$. We say that Φ_1 *and* Φ_2 *are equivalent* if for every instantiation of Γ_1 there is an equivalent instantiation of Γ_2, and vice versa.

Now to the proof of Lemma 2. Let $\Phi = (\Gamma, e, outatt, Outvars)$ be a type formula, and let $\Phi' = (\Gamma', e, outatt', Outvars)$ be its extension with the extra special attribute A. We have to show that Φ and Φ' are equivalent.

From Φ to Φ'. Let $\mathcal{I}$ be an instantiation of Γ. We have to find an equivalent instantiation $\mathcal{I}'$ of Γ'.

If $A \notin \mathcal{I}(a)$ for every $a \in Typevars$, we can simply put $\mathcal{I}'(a) := \mathcal{I}(a)$ for each type variable a, $\mathcal{I}'(B) := \mathcal{I}(B)$ for each special attribute $B \neq A$, and $\mathcal{I}'(A) := \varnothing$. In this case, it is clear that $\mathcal{I}'$ is a legal instantiation of Γ', that $\mathcal{I}(\Gamma) = \mathcal{I}'(\Gamma')$, and that the output type of Φ under $\mathcal{I}$ equals the output type of Φ' under $\mathcal{I}'$.

If $A \in \mathcal{I}(a)$ for some $a \in Typevars$, we put $\mathcal{I}'(a) := \mathcal{I}(a) - \{A\}$ for this a, and put $\mathcal{I}'(b) := \mathcal{I}(b)$ for every type variable $b \neq a$. We also put $\mathcal{I}'(B) := \mathcal{I}(B)$ for each special attribute $B \neq A$. We finally put $\mathcal{I}'(A) := \{r \mid a \in decl(r)\}$. It is clear that $\mathcal{I}'$ is a legal instantiation of Γ', and that $\mathcal{I}(\Gamma) = \mathcal{I}'(\Gamma')$. To show that the output type of Φ under $\mathcal{I}$ equals the output type of Φ' under $\mathcal{I}'$, we must show that if $a \in Outvars$, then there exists an $r \in \mathcal{I}'(A)$ such that $decl(r) \subseteq Outvars$. We will do this in Lemma 3.

From Φ' to Φ. Let $\mathcal{I}'$ be an instantiation of Γ'. We have to find an equivalent instantiation $\mathcal{I}$ of Γ.

If $\mathcal{I}'(A) = \varnothing$, then we put $\mathcal{I}(a) := \mathcal{I}'(a)$ for each type variable a, and $\mathcal{I}(B) := \mathcal{I}'(B)$ for each special attribute $B \neq A$. In this case it is clear that $\mathcal{I}'(\Gamma) = \mathcal{I}(\Gamma)$, and that the output type of Φ' under $\mathcal{I}'$ equals the output type of Φ under $\mathcal{I}$.

If $\mathcal{I}'(A) \neq \varnothing$, we know (because $\mathcal{I}'(A) \models \mathit{constraint}'(A)$) that there exists an $a \in \mathit{Typevars}$ such that $\mathcal{I}'(A) = \{r \mid a \in \mathit{decl}(r)\}$. Then we put $\mathcal{I}(a) := \mathcal{I}'(a) \cup \{A\}$, and $\mathcal{I}(b) := \mathcal{I}'(b)$ for each type variable $b \neq a$. We also put $\mathcal{I}(B) := \mathcal{I}'(B)$ for each special attribute $B \neq A$. It is now again clear that $\mathcal{I}'(\Gamma') = \mathcal{I}(\Gamma)$, and that the output type of Φ' under $\mathcal{I}'$ equals the output type of Φ under $\mathcal{I}$. ■

We still owe:

Lemma 3 *In any type formula generated by our algorithm, the following holds. Let a be a type variable in Outvars. Then there exists a relation variable r such that $a \in \mathit{decl}(r)$ and $\mathit{decl}(r) \subseteq \mathit{Outvars}$.*

Proof. By induction. The base case, $r : a \mapsto r : a$, is trivial.

For the case $e = (e_1 \cup e_2)$ we reason as follows. Let g be the symbolic solution to the system of equations. Then $\mathit{Outvars} = \bigcup g(\mathit{Outvars}_1) = \bigcup g(\mathit{Outvars}_2)$. Let $c \in \mathit{Outvars}$. Then $c \in g(a)$ for some $a \in \mathit{Outvars}_1$. By induction, we know that for some relation variable r, $a \in \mathit{decl}_1(r)$ and $\mathit{decl}_1(r) \subseteq \mathit{Outvars}_1$. This implies that $c \in \bigcup g(\mathit{decl}_1(r)) = \mathit{decl}(r)$, and that $\mathit{decl}(r) \subseteq \mathit{Outvars}$.

For the case $e = (e_1 \bowtie e_2)$ we have *Outvars* equal to $\bigcup g(\mathit{Outvars}_1) \cup \bigcup g(\mathit{Outvars}_2)$, g again being the symbolic solution. Let $c \in \mathit{Outvars}$. So, $c \in \bigcup g(\mathit{Outvars}_1)$ or $c \in \bigcup g(\mathit{Outvars}_2)$. By symmetry we may assume that $c \in \bigcup g(\mathit{Outvars}_1)$. Then $c \in g(a)$ for some $a \in \mathit{Outvars}_1$. By induction, we know that for some r, $a \in \mathit{decl}_1(r)$ and $\mathit{decl}_1(r) \subseteq \mathit{Outvars}_1$. This implies again that $c \in \mathit{decl}(r)$ and $\mathit{decl}(r) \subseteq \mathit{Outvars}$.

For the case $e = (e_1 \times e_2)$, we can use exactly the same reasoning as for $(e_1 \bowtie e_2)$, because no particular properties of the symbolic solution have been used.

The cases $e = \sigma$, ρ and $\widehat{\pi}$ are trivial because they don't change *Outvars* and *decl*. The case $e = \pi$ is trivial because it sets *Outvars* to $\varnothing$. ■

By induction on the structure of relational algebra expressions we can now prove that each case of our algorithm correctly produces a type formula

that is principal. The cases corresponding to unary operators are all proven correct in an analogous way; we treat the selection as an example below. The cases corresponding to binary operators heavily rely in addition on the correctness of our algorithm for solving systems of set equations, which we already proved correct in Section 6.

So, let $e = \sigma_{\theta(A_1,\ldots,A_n)}(e')$. Let the type formulas computed by our algoritm for e and e' be Φ and Φ', respectively. By induction, we may assume that Φ' is principal for e'; we must show that Φ is principal for e.

By Lemma 2, we may ignore step 2 of the algorithm and assume without loss of generality that for $i = 1, \ldots, n$, A_i is already in *Specattrs*$'$. More generally, we may assume that Φ differs from Φ' only in that for $i = 1, \ldots, n$,

$$\mathit{constraint}(A_i) = \mathit{constraint}'(A_i) \wedge \mathit{outatt}'(A_i)$$

and

$$\mathit{outatt}(A_i) = \mathbf{true}.$$

Now suppose $\mathcal{T} \vdash e : \tau$. We must find an instantiation $\mathcal{I}$ of Γ such that $\mathcal{T}$ equals $\mathcal{I}(\Gamma)$ and τ equals the output type of Φ under $\mathcal{I}$. Since $\mathcal{T} \vdash e : \tau$, we know that $\mathcal{T} \vdash e' : \tau$ and that for $i = 1, \ldots, n$, $A_i \in \tau$. Since Φ' is principal for e', we know furthermore that there exists an instantiation $\mathcal{I}'$ of Γ' such that $\mathcal{T}$ equals $\mathcal{I}'(\Gamma')$ and τ equals the output type of Φ' under $\mathcal{I}'$. We set the desired $\mathcal{I}$ simply equal to $\mathcal{I}'$, and verify:

- $\mathcal{I}$ is a valid instantiation of Γ: Thereto, we must check for $i = 1, \ldots, n$ that $\mathcal{I}(A_i) \models \mathit{constraint}(A_i)$, or $\mathcal{I}'(A_i) \models \mathit{constraint}'(A_i) \wedge \mathit{outatt}'(A_i)$, which is equivalent. That $\mathcal{I}'(A_i) \models \mathit{constraint}'(A_i)$ is trivial, by definition. That $\mathcal{I}'(A_i) \models \mathit{outatt}(A_i)$ is also clear, since $A_i \in \tau$ and τ equals the output type of Φ' under $\mathcal{I}'$.

- $\mathcal{T} = \mathcal{I}(\Gamma)$: This is clear, since $\mathcal{T} = \mathcal{I}'(\Gamma')$ and $\mathcal{I}(\Gamma) = \mathcal{I}'(\Gamma')$.

- τ equals the output type of Φ under $\mathcal{I}$: Since *outatt* differs from *outatt*$'$ only in that the output constraints for the A_i are loosened, the output type of Φ' under $\mathcal{I}'$, which equals τ, can only be a subset of the output type of Φ under $\mathcal{I}$. However, as every A_i is already in τ, this subset relationship cannot be a strict one, and hence the two types are indeed equal.

Conversely, suppose $\mathcal{I}$ is an instantiation of Γ, and let τ be the output type of Φ under $\mathcal{I}$. We must now show that $\mathcal{I}(\Gamma) \vdash e : \tau$. To show this,

we note that $\mathcal{I}$ is a valid instantiation of Γ' (as the attribute constraints of Γ are tighter than those of Γ'). Hence, since Φ' is principal for e', we know that $\mathcal{I}(\Gamma) = \mathcal{I}(\Gamma') \vdash e' : \tau'$, where τ' is the output type of Φ' under $\mathcal{I}$. But this output type is the same as the output type of Φ under $\mathcal{I}$; indeed, *outatt* differs only from *outatt'* on the A_i, but all A_i are members of both types anyway (for Φ' this is because $\mathcal{I}(A_i)$ satisfies $outatt'(A_i)$ by definition, and for Φ this is trivial because $outatt(A_i) = \mathbf{true}$). Hence, we have $\mathcal{I}(\Gamma) \vdash e' : \tau$. Since all the A_i are in τ, we can conclude that $\mathcal{I}(\Gamma) \vdash e : \tau$.

7.5 Complexity and typability

Since every step of the induction can be implemented in time polynomial in the size of the output of its child steps, a rough upper bound on the time complexity of our algorithm is $2^{2^{O(n)}}$. It remains open whether this complexity can be improved. Note that type formulas can be exponentially large; for example, the type formula for $r_1 \bowtie (r_2 \bowtie (\cdots \bowtie r_m) \cdots)$ uses $O(2^m)$ different type variables.

If the input expression was untypable, the algorithm will output an unsatisfiable type formula. Hence, an alternative way to check typability is to check satisfiability of the principal type formula. We do not have to wait until the end, however, to test satisfiability. In principle, as soon as an unsatisfiable attribute constraint arises during type inference, the algorithm can stop and report that the expression is untypable. This is more useful, since it tells exactly where the expression breaks down. In a practical implementation, one could do this by keeping the attribute constraints in disjunctive normal form. Doing this might actually have a better complexity than expected, since the attribute constraints generated by the algorithm have a quite special form, which might be exploited.

Note that unsatisfiable attribute constraints can only be generated in the following places:

- Step 4 of cases $\cup$ and $-$, and its adaptation for case $\times$. A simple example of a type error that will be spotted in this place is $\pi_A(r) \cup \pi_B(s)$.
- Step 3 of case σ, and its analogues for π, ρ, and $\widehat{\pi}$. A simple example of a type error that will be spotted in this place is $\sigma_{\theta(A)}(\pi_B(r))$.

Since the above-mentioned steps in the algorithm are clearly only executed if there are special attributes, we thus have:

Proposition 1 *Every expression without special attributes is typable.*

The reader might wonder about contrived examples such as

$$(r \times s) \bowtie (r \cup s),$$

which has no special attributes, but does not seem typable. However, this expression is well-typed under the type assignment by which the types of r and s are empty.

8 Polymorphic queries

Usually, a query is defined as a mapping from databases of some fixed type to relations of some fixed type. We can define a polymorphic generalization of the notion of query, to allow databases of different types as input. Fix a schema $\mathcal{S}$.

Definition 4

1. Let $\mathcal{T}$ be a type assignment on $\mathcal{S}$, and let τ be a type. A *query of type* $\mathcal{T} \to \tau$ is a mapping from databases of type $\mathcal{T}$ to relations of type τ.

2. An *input-output type family* is a *partial* function F from all type assignments on $\mathcal{S}$ to all types. We denote the definition domain of F by $\operatorname{dom} F$.

3. A *polymorphic query of type* F is a family $(Q_{\mathcal{T}})_{\mathcal{T} \in \operatorname{dom} F}$ of queries, where each $Q_{\mathcal{T}}$ is a query of type $\mathcal{T} \to F(\mathcal{T})$.

Viewed from this perspective, a type formula γ with type context Γ is, of course, nothing but a specification of an input-output family F_γ: we have $\operatorname{dom} F_\gamma = \{\mathcal{I}(\Gamma) \mid \mathcal{I} \text{ an instantiation of } \Gamma\}$, and $F_\gamma(\mathcal{I}(\Gamma))$ equals the output type of γ under $\mathcal{I}$. As a consequence, every relational algebra expression e expresses a polymorphic query of type F_γ, where γ is the principal type formula for e.

The following notion now naturally presents itself:

Definition 5 Two relational algebra expressions e_1 and e_2 are *polymorphically equivalent* if they express the same polymorphic query.

For example, the equivalence

$$\sigma_{A=B}(r \times \pi_{A,B,C}(s)) \equiv r \times \sigma_{A=B}\pi_{A,B,C}(s)$$

is polymorphic, but the equivalence

$$\pi_A(r \bowtie \pi_{A,B}(s)) \equiv \pi_A(r \bowtie s)$$

is not, as it is only valid under a type assignment $\mathcal{T}$ such that $\mathcal{T}(r) \cap \mathcal{T}(s)$ is a subset of $\{A, B\}$.

We are now weaponed to return to the issue of non-redundancy already touched upon at the end of Section 2.

Proposition 2 *1. There is no expression not using $\bowtie$ that is polymorphically equivalent to $r \bowtie s$. We say that $\bowtie$ is* polymorphically non-redundant. *The same holds for the operator $\times$.*

2. There is no expression not using π that is polymorphically equivalent to $\pi_A(r)$. So, also π is polymorphically non-redundant. The same holds for the operator $\widehat{\pi}$.

Proof. Any expression e polymorphically equivalent to $r \bowtie s$ must have principal type

$$\begin{array}{l} r : a_1a_2 \\ s : a_2a_3 \end{array} \mapsto e : a_1a_2a_3.$$

Inspecting the principal type inference algorithm, we see that a type formula where *Outvars* contains the union of $decl(r)$ and $decl(s)$, where the latter two sets are different and have a non-empty intersection, can only be produced in the case of $\bowtie$. An analogous argument deals with $\times$.

As for $\pi_A(r)$, any polymorphically equivalent expression e must have principal type

$$\begin{array}{lcl} r : a & \mapsto & e : \varnothing \\ A : r & & A : \textbf{true} \end{array}.$$

Inspecting the principal type inference algorithm, we see that a type formula where *Outvars* is made empty, depending on some special attribute, can only be produced in the case of π. An analogous argument deals with $\widehat{\pi}$. ■

We can also show polymorphic inexpressibility results for the full language. For example:

Proposition 3 *The semijoin $r \ltimes s$ is not* polymorphically *expressible in the standard relational algebra.*

Proof. Suppose e is an expression polymorphically equivalent to $r \ltimes s$. The principal type of e must be

$$\begin{array}{l} r : a_1 a_2 \\ s : a_2 a_3 \end{array} \mapsto e : a_1 a_2.$$

Since there are no special attributes, the operators σ, ρ, $\widehat{\pi}$, and π cannot occur in e, except for $\pi_\varnothing$ (projection on the empty sequence of attributes). Now consider the type assignment $\mathcal{T}$ on $\{r, s\}$ given by $\mathcal{T}(r) = \{A, B\}$ and $\mathcal{T}(s) = \{B, C\}$, and the database $\mathbf{D}$ of type $\mathcal{T}$ defined by $\mathbf{D}(r) = \{[A : x, B : y], [A : u, B : v]\}$ and $\mathbf{D}(s) = \{[B : y, C : z]\}$. Given $\mathcal{T}$, the type of e is $\{A, B\}$. Using the above knowledge of e, we can see that in the value of e on $\mathbf{D}$, either $[A : x, B : y]$ and $[A : u, B : v]$ both occur, or none of them occurs. However, this is in contradiction with the fact that e is equivalent to $r \ltimes s$. Hence, e does not exist. ■

9 Concluding remarks

We have seen in the previous section that classical "derived" operators of the standard relational algebra can become primitive in the polymorphic setting. The same holds for many other such operators. Note that it is actually easy to extend our type inference algorithm to include semijoin and similar operators, so Proposition 3 should not be misinterpreted as a negative result. Rather, it indicates that the new issue arises as to how a basic polymorphic query language should be designed. This is an interesting direction for further work.

As already mentioned in the Introduction, other obvious directions for further work include (i) applying type inference in practice to SQL rather than to the relational algebra; (ii) developing type inference in the context of semi-structured data models rather than the relational data model; or (iii) to do the same for object-oriented query languages such as OQL. When moving to the OO context, one has to deal with the additional subtilities created by inheritance and subtyping. Current research in programming languages is giving these issues considerable attention.

We have also ignored types on the level of individual attribute values, although such types are almost always present in practice, e.g., in SQL. For

example, for $\sigma_{A=\text{“John”}}(r)$ to be well-typed it suffices for us that the type of r has an A-attribute. However, in reality, A must in addition be of type string. Incorporating types on the attribute value level only has an effect on the special attributes of an expression; it has no effect on its polymorphic basis (recall the notion of polymorphic basis from Section 4). Hence, a type inference algorithm can still be based on solving systems of set equations. When conjugating two type contexts, however (recall Section 7.1.2), a unification on the value types associated to the special attributes has to be performed. A similar unification is induced by the natural join operator. Moreover, in the case of the selection operator, the selection predicate (which in our approach has remained abstract) will perform certain operations on certain special attributes, which will induce certain constraints on the value types associated to these attributes. In general, if the programming language in which we write selection predicates has a unification-based type system, then we can simply activate type inference for this system at the appropriate places.

Acknowledgment

We thank Serge Abiteboul, who suggested the idea of type inference for relational algebra to the second author many years ago; Didier Rémy and Limsoon Wong, for helpful conversations; and Julien Forest and Veronique Fischer, who implemented preliminary versions of the algorithm.

References

[1] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers*. Addison-Wesley, 1986.

[2] P. Buneman, S. Davidson, M. Fernandez, and D. Suciu. Adding structure to unstructured data. In F. Afrati and Ph. Kolaitis, editors, *Database Theory—ICDT'97*, volume 1186 of *Lecture Notes in Computer Science*, pages 336–350. Springer, 1997.

[3] P. Buneman, S. Davidson, G. Hillebrand, and D. Suciu. A query language and optimization techniques for unstructured data. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, issue 25:2 of *SIGMOD Record*, pages 505–516. ACM Press, 1996.

[4] H. Garcia-Molina, Y. Papakonstantinou, D. Quass, A. Rajaraman, Y. Sagiv, J. Ullman, V. Vassalos, and J. Widom. The TSIMMIS approach to mediation: data models and languages. *Journal of Intelligent Information Systems*, 8(2):117–132, 1997.

[5] P. Giannini, F. Honsell, and S. Ronchi della Rocca. Type inference: some results, some problems. *Fundamenta Informaticae*, 19:87–125, 1993.

[6] C.A. Gunter and J.C. Mitchell, editors. *Theoretical Aspects of Object-Oriented Programming*. MIT Press, 1994.

[7] J.R. Hindley. *Basic Simple Type Theory*. Cambridge University Press, 1997.

[8] J. Melton. *Understanding SQL's Stored Procedures*. Morgan Kaufmann, 1998.

[9] J.C. Mitchell. *Foundations for Programming Languages*. MIT Press, 1996.

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

[11] A. Ohori, P. Buneman, and V. Breazu-Tannen. Database programming in Machiavelli—a polymorphic language with static type inference. In *Proceedings of the 1989 ACM SIGMOD International Conference on the Management of Data*, issue 18:2 of *SIGMOD Record*, pages 46–57. ACM Press, 1989.

[12] D. Rémy. Type inference for records in a natural extension of ML. In Gunter and Mitchell [6], pages 67–96.

[13] D. Rémy. Typing record concatenation for free. In Gunter and Mitchell [6], pages 351–372.

[14] D. Stemple et al. Exceeding the limits of polymorphism in database programming languages. In F. Bancilhon, C. Thanos, and D. Tsichritzis, editors, *Advances in Database Technology—EDBT'90*, volume 416 of *Lecture Notes in Computer Science*, pages 269–285. Springer-Verlag, 1990.

[15] J. Tiuryn. Type inference problems: a survey. In B. Rovan, editor, *Mathematical Foundations of Computer Science*, volume 452 of *Lecture Notes in Computer Science*, pages 105–120, 1990.

[16] J.D. Ullman. *Elements of ML Programming*. Prentice-Hall, 1998.

[17] S. Vansummeren. An implementation of polymorphic type inference for the relational algebra, written in the programming language ML. Master's thesis, University of Maastricht, 2001.