Algorithm and Complexity of the Unification Problem of a Polymorphic Attribute-based Type System

Ken Q. Pu

University of Toronto, Toronto ON, Canada
kenpu@cs.toronto.edu

Abstract. We introduce a polymorphic attribute-based type system capable of expressing type information of various query constructs. It extends the existing record-type systems by introducing attribute variables to model polymorphic attribute names. Such extension permits modeling query constructs of various languages as typed functional symbols.
Type-checking and type-inference of database queries are reduced to unification of type-expressions in our type system. We first establish that the unification is NP-complete. A complete unification algorithm in EXPTIME is presented. Finally, we characterize a useful tractable sub-class of unification for which the complete unification algorithm terminates in PTIME.

1 Introduction

Types are an essential ingredient of any modern programming language because of the universal fact that they reduce programming errors and make programs safer. In query language design, type-theoretic ideas have been proven to be quite useful [3, 7, 14]. By assigning types to query expressions, one can perform static analysis including *type-checking* and *type-inference* on database queries as one can do to programs. The problem of *type-checking* is to ensure that all applications of operators are valid with respect to the type system, while the more generalized problem of *type-inference* is to infer some unspecified type information of an expression so that it is validly typed.

In this paper, we introduce a polymorphic attribute-based type system that can be used to express type information of a wide range of query constructs from functional query languages [2, 3] and nested relational algebra with aggregation [1]. In addition to the classical *row-variables* in record-types [13, 15], we enrich the type system with *attribute-variables* to model unknown attribute-names in polymorphic types. This is necessary to model certain database query constructs as higher-order functions. Consider the simple rename-operator ρ of relational algebra which renames an attribute of its input relation. In order to represent it as a polymorphic function, we use attribute variables as follows[1]:

$$\rho : \mathtt{set}(\{u_1 : \boldsymbol{x}_1,\ \boldsymbol{x}_2\}) \to \mathtt{set}(\{u_2 : \boldsymbol{x}_1,\ \boldsymbol{x}_2\}),$$

[1] We formally define the type-expression in later sections.

where u_i are the attribute-variables, and $\boldsymbol{x}_i$ the row-variables. It renames the attribute u_1 to attribute u_2. Note that the classical row-variables [13, 15, 3] cannot express the rename-operator. Another example is the nest-operator of nested relational algebra:

$$\mathsf{nest} : \texttt{set}(\{\boldsymbol{x}_1,\ \boldsymbol{x}_2\}) \to \texttt{set}(\{\boldsymbol{x}_1,\ u_1{:}\texttt{set}(\{\boldsymbol{x}_2\})\}).$$

The nest-operator creates a new attribute u_1 whose type is another relation $\texttt{set}(\boldsymbol{x}_2)$. We can also apply the type system to functional query languages [2, 3]. Consider the map-operator that operates on records:

$$\mathsf{map} : (\{\boldsymbol{x}_1,\ \boldsymbol{x}_2\} \to \{\boldsymbol{x}_3\}) \to (\{u_1{:}\texttt{list}(\{\boldsymbol{x}_1\}),\ \boldsymbol{x}_2\} \to \{u_2{:}\texttt{list}(\{\boldsymbol{x}_3\})\}).$$

It is a second-order function. With an input function $f : \{\boldsymbol{x}_1, \boldsymbol{x}_2\} \to \boldsymbol{\{}x_3\}$, $\mathsf{map}f$ iterates on a list $\texttt{list}(\boldsymbol{x}_1)$ identified by some attribute u_1, and the output consists of an attribute of the type $\texttt{list}(\boldsymbol{x}_3)$.

Type-checking and type-inference of query languages reduce to unification of type-expressions in our type system. Given two partially specified types, the *unification problem* is to solve for the unspecified variables in the two type expressions such that they can be equated.

We prove that the unification problem of our type system is NP-complete in general. A complete unification algorithm is presented that runs in EXPTIME. Finally we identify a useful tractable sub-class for which the unification algorithm terminates in PTIME.

2 Related Work

Several works [3, 7, 14] study the issue of assigning type information to relational query languages, and their related type-checking and type-inference problems. Buneman and Ohori [3] studies the problem of type-checking and inference for a proposed query language based on functional programming, while Bussche *et. al* [7] and Vansummeren [14] study a type system for the classical relational algebra, and the associated type-inference problem. The type-inference algorithms for the functional query language and relational algebra are specific to the respective query languages of interest.

Our motivation is to introduce a polymorphic attribute-based type system flexible enough such that the query constructs found in these languages can be modeled as second-order functional symbols with types specified using our type system. This means that we are able to perform type-checking and type-inference in a query language-independent way using unification of type-expressions. We omit the details of how unification is used to perform type-checking and inference as it is the canonical approach for functional programming [10, 6, 11].

It is particularly convenient and natural to use record-based types in relational databases since often a relation is seen as a collection of records with named fields. Record-types in the context of functional programming languages have been studied [15, 13, 4]. The proposed record-type systems allow the use of row-variables to model extensible records. We extend their work by also introducing attribute-variables. As seen above, the attribute-variables enable representation of query operators of various query languages.

Record-types have also been used in in computational linguistics, where they are referred to as *feature structures*. Unification of feature structures has been well-studied [9, 5]. However, for feature structures, the usage of row-variables are limited, so efficient unification algorithms exist. In contrast, we show that our generalization of allowing attribute-variables and flexible use of row-variables makes the unification problem intractable.

3 The type system

The type system describes attribute-based records.

Definition 1 (Type expressions). *Let $\mathbf{T}_0$ be a set of atomic types, $\mathbf{A}$ a set of constants representing attribute names, and $\mathbf{V}^{\mathrm{attr}} = \{u_1, u_2, \dots\}$ and $\mathbf{V}^{\mathrm{row}} = \{\boldsymbol{x}_1, \boldsymbol{x}_2, \dots\}$ are sets of attribute- and row-variable names.*

Let $\mathbf{T}$ denote the set of type-expressions, defined recursively as follows.

A **type-expression** *$t \in T$ is a set of field-expressions $f_1, f_2, \cdots$ and row-variables $\boldsymbol{x}_1, \boldsymbol{x}_2, \cdots$, written $t = \{f_1, f_2, \cdots, \boldsymbol{x}_1, \boldsymbol{x}_2, \cdots\}$. It is required that the fields do not have duplicate attribute-names.*

A **field-expression** *f is an attribute-type pair of the form $a : t$ where $a \in \mathbf{A} \cup \mathbf{V}^{\mathrm{attr}}$, and t a type-expression or a collection-expression. A* **collection-expression** *is of the form:* $\mathtt{set}(t)$, $\mathtt{bag}(t)$ *or* $\mathtt{list}(t)$ *where $t \in \mathbf{T}$.*

Therefore, a type-expression $t \in \mathbf{T}$ is a record-type, characterized by a set of attributes and their types.

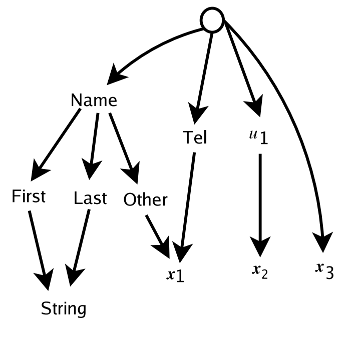
Example 1. Consider the following type-expression:

$$t = \{\mathsf{Name} : \{\mathsf{First} : \mathtt{String}, \mathsf{Last} : \mathtt{String}, \mathsf{Other} : \boldsymbol{x}^1\}, \mathsf{Tel} : \boldsymbol{x}^1, u^1 : \boldsymbol{x}^2, \boldsymbol{x}^3\}$$

We can write the attributes vertically as show in Figure 1(a), or see it as a DAG as Figure 1(b).

$$t = \left[\begin{array}{l} \mathsf{Name} : \left[\begin{array}{l} \mathsf{First} : \mathtt{String} \\ \mathsf{Last} : \mathtt{String} \\ \mathsf{Other} : \boldsymbol{x}^1 \end{array}\right. \\ \mathsf{Tel} : \boldsymbol{x}^1 \\ u^1 : \boldsymbol{x}^2 \\ \boldsymbol{x}^3 \end{array}\right.$$

(a) A type-expression



(b) A DAG representation

Fig. 1. Representations of a type-expression

Definition 2 (Substitutions and instances). *A substitution θ is a mapping on the variables $\mathbf{V}^{\mathrm{attr}} \cup \mathbf{V}^{\mathrm{row}}$ such that:*

- *for all $u \in \mathbf{V}^{\mathrm{attr}}$, $\theta(u) \in \mathbf{A} \cup \mathbf{V}^{\mathrm{attr}}$, and*
- *for all $\boldsymbol{x} \in \mathbf{V}^{\mathrm{row}}$, $\theta(\boldsymbol{x}) \in \mathbf{T}$.*

A substitution θ extends to type-expression in $\mathbf{T}$. Given a type t, $\theta(t)$ is the type-expression obtained by replacing all attribute-variables u in t by $\theta(u)$, and $\boldsymbol{x}$ in t by fields (and possibly other row-variables) in $\theta(\boldsymbol{x})$.

Given some θ, we refer to $\theta(t)$ as an instance of t, provided that $\theta(t)$ is a valid type-expression in $\mathbf{T}$.

Given two substitutions, θ_1 and θ_2, we say that $\theta_1 \leq \theta_2$ if there exists another substitution λ such that $\theta_2 = \lambda \circ \theta_1$.

Example 2. Recall the type-expression t in Example 1. Let

$$\theta_1 = \{\boldsymbol{x}^2 \mapsto \left[\begin{array}{l}\mathsf{City} : \mathtt{String} \\ \mathsf{Street} : \mathtt{String}\end{array}\right., \quad \boldsymbol{x}^3 \mapsto \emptyset, \quad u^1 \mapsto \mathsf{Address}\}$$

Then

$$\theta_1(t) = \left[\begin{array}{l}\mathsf{Name} : \left[\begin{array}{l}\mathsf{First} : \mathtt{String} \\ \mathsf{Last} : \mathtt{String} \\ \mathsf{Other} : \boldsymbol{x}^1\end{array}\right. \\ \mathsf{Tel} : \boldsymbol{x}^1 \\ \mathsf{Address} : \left[\begin{array}{l}\mathsf{City} : \mathtt{String} \\ \mathsf{Street} : \mathtt{String}\end{array}\right.\end{array}\right.$$

Consider $\theta_2 = \{u^1 \mapsto \mathsf{Name}\}$. The substitution $\theta_2(t)$ is undefined since instantiating the attribute-variable u^1 to Name produces duplicate attribute names among sibling fields.

4 Unification and its complexity

Definition 3. *A type-equation is simply a pair (t, t') of type expressions, written as $t \sim t'$. Let S be a set of type-equations $\{t_i \sim t'_i : i \leq n\}$. A unifier of S is a substitution θ such that for all equations $t_i \sim t'_i$ in S, $\theta(t_i) = \theta(t'_i)$. The set S is unifiable if there exists such a unifier.*

The set of unifiers of S is denoted by $\mathbf{U}(S)$, and the minimal unifiers, i.e., the most general unifiers of S by $\min \mathbf{U}(S)$.

Our unification problem is distinct in several aspects:

- The order of fields do not matter. This means that the syntactic unification algorithms [12] do not apply.
- There can be multiple occurrence of a row-variable in an equation, and equations can share row-variables, making our problem different from unification of feature structures [5].
- Finally, we make use of attribute-variables, which has not been considered by literature dealing with record-types [15, 13, 4].

First we show that the flexibility of our type system makes the unification problem intractable. In fact, the intractability results from the unrestricted usage of either row- or attribute-variables alone.

Theorem 1. *The unification problem is NP-hard. It remains NP-hard even if the type-expressions only make use of row-variables or only make use of attribute-variables.*

Proof (Sketch). First we reduce the 3-SAT problem to unification of type-equations that only make use of attribute-variables. Consider an instance of 3-SAT problem: $\{C_i\}$ where each C_i is a clause of three literals and each literal is a boolean variable or its negation. For each boolean variable x, introduce a type-equation $\{u_x : \mathsf{A}, u_{\bar{x}} : \mathsf{A}\} \sim \{\mathsf{T} : \mathsf{A}, \mathsf{F} : \mathsf{A}\}$, where A is some atomic type. For each clause $C_i = \{t_{i1}, t_{i2}, t_{i3}\}$, introduce a type-equation

$$\begin{bmatrix} \mathsf{c}_1 : \{u_{i1} : \mathsf{A}, u_{i2} : \mathsf{A}\} \\ \mathsf{c}_2 : \{u_{i3} : \mathsf{A}, \mathsf{F} : \mathsf{A}\} \end{bmatrix} \sim \begin{bmatrix} v_1 : \{\mathsf{T} : \mathsf{A}, v_3 : \mathsf{A}\} \\ v_2 : \{v_4 : \mathsf{A}, v_5 : \mathsf{A}\}, \end{bmatrix}$$

where v_i are fresh attribute variables, and u_{ij} are defined as u_x if $t_{ij} = x$, and $u_{\bar{x}}$ if $t_{ij} = \bar{x}$.

It follows that the equations are unifiable if and only if $\bigwedge\{C_i\}$ are satisfiable.

Next we show that monotone one-in-three 3SAT problem [8] is reducible to unification of equations that only use row-variables. For each boolean variable x, have $\{\boldsymbol{y}_x, \boldsymbol{y}_{\bar{x}}\} = \{\mathsf{c} : \mathsf{A}\}$. For each clause $C_i = \{x_{i1}, x_{i2}, x_{i3}\}$, have $\{\boldsymbol{y}_{i1}, \boldsymbol{y}_{i2}, \boldsymbol{y}_{i3}\} = \{\mathsf{c} : \mathsf{A}\}$. It follows that the equations are unifiable if and only if that the instance of monotone one-in-three 3SAT problem has a solution. $\square$

5 A complete unification algorithm

In this section, we present a complete unification algorithm for our type system. The algorithm is based on the equivalence refinement algorithm for syntactic unification. The algorithm operates on the directed acyclic graph (DAG) representation of type-expressions, and it incrementally builds an equivalence of nodes of the DAG. In the case of syntactic unification, the algorithm can be substantially optimized to run in linear time [12]. However, in our case it runs in EXPTIME, which is expected due to the intractability result of Theorem 1.

Aside from serving as a complete unification algorithm for our type-system, its analysis leads to two important results.

- The unification problem is in NP, thus is NP-complete.
- We can identify a tractable class of unification problems for which the algorithms run in PTIME. Furthermore, w show that this class is maximal in that any relaxation of this class leads to intractability.

5.1 Unification of DAG's and the unification relation

As shown in Example 1, a type-expression can be naturally represented as a DAG. We denote the nodes of a DAG D by $\mathbf{N}(D)$, nodes that are labeled by constants (attribute names or primitive types) by $\mathbf{N}^{\mathrm{CON}}(D)$, the nodes labeled by attribute-variables by $\mathbf{N}^{\mathrm{VATTR}}(D)$ and nodes labeled by row-variables by $\mathbf{N}^{\mathrm{VROW}}(D)$. We drop the argument D if the DAG of interest is clear from the context. The labeling function is written as $\ell_D : n \mapsto \ell_D(n)$.

One can instantiate a DAG D by a substitution θ: if D represents the type expression t, then θD is a DAG that represents $\theta(t)$. In terms of the graph representation, instantiating a DAG involves:

– Each node labeled by attribute-variable u in $\mathbf{N}^{\text{VATTR}}$ is relabeled by $\theta(u)$.
– Each node labeled by row-variable $\boldsymbol{x}$ in $\mathbf{N}^{\text{VROW}}$ is either expanded to possibly multiple sub-graphs, $\theta(\boldsymbol{x})$, or deleted if $\theta(\boldsymbol{x}) = \emptyset$.
– Any common subgraphs are merged.

Without loss of generality, we can assume only one equation to be unified. In terms of DAG representation of s and t, the unification problem can be cast as the following.

Definition 4. *The unification problem of nodes n_s, n_t in a DAG D is defined as follows: Given two nodes $n_s, n_t \in \mathbf{N}(D)$, find a substitution θ such that $\theta(D(n_s)) = \theta(D(n_s))$.*

For the presentation of the algorithm, we introduce the *node-variables* $\mathbf{V}^{\text{node}}$, and nodes that are labeled by them, $\mathbf{N}^{\text{VNODE}}(D)$. We write node-variables as $\dot{x}, \dot{y}, \cdots$. Nodes in $\mathbf{N}^{\text{VNODE}}$ must only occur uniquely at the leaf level of D. It is required that an instantiation can only replace a node-variable $\dot{x}$ by a type with at most one attribute, or a primitive type. Equivalently, a node in $\mathbf{N}^{\text{VNODE}}(D)$ can only be replaced by a subgraph with a distinguished root node as oppose to the general row-variables which can be replaced by multiple sub-graphs (or none at all).

Definition 5. *Let D be an DAG, with nodes $\mathbf{N}$. A unification relation is an equivalence relation $\mathcal{E}$ defined on the nodes $\mathbf{N}$ satisfying the following conditions:*

– DISTINCT-SIBLINGS*: No two siblings are equivalent.*
– HOMOGENEITY*: A subset $M \subseteq \mathbf{N}$ of nodes is called homogeneous if $(\forall n, n' \in M \cap \mathbf{N}^{\text{CON}})\ \ell_D(n) = \ell_D(n')$. We require*

$$(\forall n \in N)\ \ [n]_{\mathcal{E}} \text{ is homogeneous, and}$$
$$(\forall u \in \mathbf{V}^{\text{attr}})\ \ \left(\bigcup \left\{[n]_{\mathcal{E}} : n \in \ell_D^{-1}(u)\right\}\right) \text{ is homogeneous.}$$

– CONGRUENCE*: For all $n_1, n_2 \in \mathbf{N}(D)$ such that $n_1, n_2 \notin \mathbf{N}^{\text{VNODE}}(D) \cup \mathbf{N}^{\text{VROW}}(D)$, $n_1 \equiv_{\mathcal{E}} n_2 \implies \downarrow_D(n_1)/\mathcal{E} = \downarrow_D(n_2)/\mathcal{E}$ where $\downarrow_D(n)$ are the children nodes of n in D, and $\downarrow_D(n_i)$ the equivalence classes of $\mathcal{E}$ containing $\downarrow_D(n_i)$.*
– ACYLICITY*: If n_1 is an ancestor of n_2, then $n_1 \not\equiv_{\mathcal{E}} n_2$.*

A unification relation $\mathcal{E}$ on a DAG D is complete if for all equivalent pairs of nodes $n_1 \equiv_{\mathcal{E}} n_2$ such that n_1 and n_2 are not node- nor row-variables, $(\forall m_1 \in\downarrow(n_1))(\exists m_2 \in\downarrow(n_2)) m_1 \equiv_{\mathcal{E}} m_2$. Otherwise, $\mathcal{E}$ is partial.

Proposition 1. *Let D be a DAG without row variables. Then $n_1, n_2 \in N$ are unifiable if and only if there exists a complete unification relation $\mathcal{E}$ such that $n_1 \equiv_{\mathcal{E}} n_2$.*

The general approach of our algorithm is to successively expand the row-variable in $\mathbf{N}^{\text{VROW}}(D)$ to a set of fresh node-variables and compute unification relations. Therefore, the solution to the problem of unifying nodes n_s, n_t in a DAG D is pair $(\mathcal{E}, \theta)$ where θ is a substitution that only expands row-variables to fresh node-variables, and $\mathcal{E}$ a unification relation on the DAG θD such that $n_1 \equiv_{\mathcal{E}} n_2$. We say that $(\mathcal{E}, \theta)$ is a complete solution if $\mathcal{E}$ is complete in θD, otherwise $(\mathcal{E}, \theta)$ is a partial solution. There is always a partial solution to the unification of nodes n_s and n_t in D: $(\mathcal{E}_0, D)$ where $\mathcal{E}_0 = \{\{n_s, n_t\}\}$. The algorithm tries to extend this initial partial solution by means of local-extensions defined below until a complete solution is found or no further extensions can be made.

Definition 6. *Let $(\mathcal{E}, D)$ be such that $\mathcal{E}$ is a partial unification relation on the DAG D, An extension of $(\mathcal{E}, D)$ is a pair $(\mathcal{E}', \theta D)$ where θ is a substitution that only expands row-variable in $\mathbf{N}^{\text{VROW}}(D)$ with fresh node-variable, and $\mathcal{E}'$ a unification relation on θD such that $(\forall n, n' \in \mathbf{N}(D))\ n \equiv_{\mathcal{E}} n' \implies n \equiv_{\mathcal{E}'} n'$.*

We say that $(\mathcal{E}', \theta D)$ is an (n_1, n_2)-extension of $(\mathcal{E}, D)$ for some $n_1, n_2 \in \mathbf{N}^{\text{CON}}(D) \cup \mathbf{N}^{\text{VATTR}}(D)$ if $n_1 \equiv_{\mathcal{E}} n_2$, and $\equiv_{\mathcal{E}'} = (\equiv_{\mathcal{E}} \cup R)^*$ for some relation $R \subseteq \downarrow_{\theta D}(n_1) \times \downarrow_{\theta D}(n_2)$.[2] The extension $(\mathcal{E}', \theta D)$ is a complete (n_1, n_2)-extension if $\mathcal{E}'$ is complete at n_1 and n_2. A (complete) local extension of $(\mathcal{E}, D)$ is simply a (complete) (n_1, n_2)-extension for some $n_1, n_2 \in \mathbf{U}(\mathcal{E})$.

Let $(\mathcal{E}_1, \theta_1 D)$ and $(\mathcal{E}_2, \theta_2 D)$ both be (n_1, n_2)-extensions of $(\mathcal{E}, D)$, we say $(\mathcal{E}_1, \theta_1 D) \leq (\mathcal{E}_2, \theta_2 D)$ if $(\mathcal{E}_2, \theta_2 D)$ is an (n_1, n_2)-extension of $(\mathcal{E}_1, \theta_1 D)$. Therefore, one may speak of minimal and complete (n_1, n_2)-extensions.

5.2 The unification algorithm

The algorithm is shown in Figure 2. The function local_extensions$(\mathcal{E}, D, n_1, n_2)$ computes all the local-extensions of a partial solution $(\mathcal{E}, D)$ at nodes (n_1, n_2), and the function unification(n_s, n_t, D) computes all minimal solutions of unifying nodes n_s and n_t in the DAG D.

5.3 Analysis of the algorithm

Proposition 2. *Let $(\mathcal{E}, \theta)$ be a minimal solution to the unification problem of n_s, n_t in D. Then, for all row-variables $\boldsymbol{x}$ in D, $|\theta(\boldsymbol{x})| \leq |\mathbf{N}(D)|$. Therefore, the unification problem is in NP.*

Proof (Sketch). First we show by induction on the height of the DAG, that any complete solution $(\mathcal{E}, \theta)$ to the unification of two nodes (n_s, n_t) in D can be obtained by a finite sequence of minimal local-extensions from the initial partial solution of $(\mathcal{E}_0, D)$ where $\mathcal{E}_0 = \{\{n_s, n_t\}\}$.

Next, we show that the node-variables created at each minimal local-extension is equivalent to some nodes in $\mathbf{N}^{\text{VATTR}}(D) \cup \mathbf{N}^{\text{CON}}(D)$. By the DISTINCT-SIBLING

[2] We think of $\equiv_{\mathcal{E}}$ as a binary relation on $\mathbf{N}(D)$. Therefore, $\equiv_{\mathcal{E}} \cup R$ is another binary relation, and $(\equiv_{\mathcal{E}} \cup R)^*$ is the transitive, reflexive and symmetric closure of $(\equiv_{\mathcal{E}} \cup R)$.

```
Function local_extensions($\mathcal{E}, D, n_1, n_2$)
Let $A_1 = \downarrow_D(n_1)$, $A_2 = \downarrow_D(n_2)$, $\mathbb{E} = \emptyset$.
For Each $R \subseteq A_1 \times A_2$, do
    For Each $\boldsymbol{x} \in A_1 \cup A_2$, do
        Let $k =$ number of edges of $R$ connected to $\boldsymbol{x}$.
        If $A_1 \cap \mathbf{N}^{\text{VROW}}(D) \neq \emptyset$ and $A_2 \cap \mathbf{N}^{\text{VROW}}(D) \neq \emptyset$, then
            $\theta(\boldsymbol{x}) = \{\boldsymbol{x}, \dot{x}_1, \ldots, \dot{x}_k\}$ where all $\dot{x}_i$
        Else
            $\theta(\boldsymbol{x}) = \{\dot{x}_1, \ldots, \dot{x}_k\}$ where all $\dot{x}_i$ are fresh.
        End if.
        Let $D' = \theta D$ and $\mathcal{E}' = (\mathcal{E} \cup R)^*$.
        If $\mathcal{E}'$ is a unification relation of $D'$ then
            let $\mathbb{E} = \mathbb{E} \cup \{(\mathcal{E}', D')\}$.
        End If
    End For
End For
Return $\mathbb{E}$.
```

```
Function unification($n_s$, $n_t$, $D_0$)
Let $\mathcal{E}_0 = \{\{n_s, n_t\}\}$
If $\mathcal{E}_0$ is complete, then return $\{\mathcal{E}_0\}$ and exit.
Let $\mathbb{E}^{\text{extend}} = \{(\mathcal{E}_0, D_0)\}$, $\mathbb{E}^{\text{final}} = \emptyset$.
Loop while $\mathbb{E}^{\text{extend}} \neq \emptyset$
    Pick $(\mathcal{E}, D) \in \mathbb{E}^{\text{extend}}$, and $n_1, n_2 \in \mathbf{N}^{\text{CON}}(D) \cup \mathbf{N}^{\text{VATTR}}(D)$
        such that $n_1 \equiv_{\mathcal{E}} n_2$, $\mathcal{E}$ incomplete at $n_1$ or $n_2$.
    Let $\mathbb{E} =$ local_extensions($\mathcal{E}, D, n_1, n_2$).
    If $\mathbb{E} = \emptyset$ then delete $(\mathcal{E}, D)$ from $\mathbb{E}^{\text{extend}}$,
    Else replace $(\mathcal{E}, D)$ with elements of $\mathbb{E}$ in $\mathbb{E}^{\text{extend}}$.
    $\mathbb{E}^{\text{final}} = \mathbb{E}^{\text{final}} \cup \{(\mathcal{E}, D) \in \mathbb{E}^{\text{extend}} : \mathcal{E}$ is complete.$\}$
    $\mathbb{E}^{\text{extend}} = \{(\mathcal{E}, D) \in \mathbb{E}^{\text{extend}} : \mathcal{E}$ is incomplete.$\}$
End loop
Return $\mathbb{E}^{\text{final}}$.
```

Fig. 2. The unification algorithm

property of unification relations, for all row-variables $\boldsymbol{x}$, no two node variables $\dot{x}, \dot{y} \in \theta(\boldsymbol{x})$ can be equivalent to the same node. Therefore, $|\theta(\boldsymbol{x})| \leq |\mathbf{N}^{\text{VATTR}}(D) \cup \mathbf{N}^{\text{CON}}(D)| \leq |\mathbf{N}(D)|$.

This proves that the size of the solution is polynomially bounded by the size of the graph D. Thus, the unification problem is in NP. □

Together with Theorem 1, we conclude the following.

Theorem 2. *Unification of type-expression in our type-system is NP-complete.*

Proposition 3. *The number of successive local-extensions, namely the number of iterations of unification(n_s, n_t, D), is bounded by $|N(D)|^2$.*

Proof (Sketch). Using Proposition 2, we can bound the number node-variables required in a complete solution of a unification problem: $|\mathbf{N}^{\text{VROW}}(D)| \cdot |N(D)| \leq |N(D)|^2$. Since each successive local-extension either produces at least one node-variable or eliminate a row-variable, one only needs to compute at most $|N(D)|^2$ successive local-extensions. □

Proposition 4. *The worst-time complexity of the unification algorithm is EXPTIME.*

Proof (Sketch). Let $N = \mathbf{N}(D)$. There are at most $\mathcal{O}\left(2^{N^2}\right)$ local-extensions, and since the length of chain of successive local-extensions is bounded by N^2, the complexity of unification is $\mathcal{O}\left(2^{N^4}\right) \in$ EXPTIME. □

6 A tractable case

In this section, we identify a tractable case of the unification problem.

Definition 7. *The ambiguity of a node is the number of children that are either attribute-variable or row-variable nodes:*

$$am(n) = \big| \downarrow(n) \cap (\mathbf{N}^{\text{VATTR}}(D) \cup \mathbf{N}^{\text{VROW}}(D))\big|.$$

The ambiguity of a DAG is its maximum node ambiguity:

$$am(D) = \max\{am(n) : n \in \mathbf{N}(D)\}.$$

A node n (DAG D) is ambiguous if $am(n) > 1$ ($am(D) > 1$), otherwise, we say that it is unambiguous.

Theorem 3. *A unification problem with an unambiguous DAG*

- *has a unique minimal solution, and*
- *can be solved in* PTIME.

Proof (Sketch). We first show that, given a unification problem of n_s and n_t in an unambiguous DAG D, every partial solution $(\mathcal{E}_i, D_i)$ in the sequence of successive local-extensions has a unique minimal and complete (n_1, n_2)-extension.

Therefore, in the unification algorithm, the bound on the size of the output of local_extensions collapses from $\mathcal{O}(2^{N^2})$ to 1. As a result, the solution tree explored by unification collapses to a chain of successive local-extensions with length of at most N^2, and the function unification terminates in PTIME with at most one solution. □

The tractable case stated above includes known tractable unifications of syntactic unification, unification of extensible records and unification of feature structures. The difference is that we still consider the occurrence of attribute-variables. Unambiguous unification is the most relaxed tractable class of unification of type-expressions as we show in the following proposition that any relaxation leads to NP-hardness.

Proposition 5. *The unification problem of two nodes in a DAG D is still NP-hard if $am(D) = 2$. Furthermore, it remains NP-hard in the following cases: $am(D) = 2$, and for each ambiguous node n,*

- *the two variable nodes in $\downarrow n$ are both attribute-variable nodes.*
- *the two variable nodes in $\downarrow n$ are both row-variables,*
- *the two variable nodes in $\downarrow n$ consists of one attribute-variable node and one row-variable.*

Proof (Sketch). Observe that in the proof of Theorem 1, the type-expressions used are with ambiguity of at most 2. Therefore, we already showed the NP-hardness of the first two cases.

One can reduce the SAT problem to unification of type-expressions whose ambiguous nodes consists of an attribute-variable and a row-variable. Thus it is also NP-hard. □

7 Conclusion and Future Work

We have presented an attribute-based type system that is suitable for expressing type information of query constructs found in several query languages including relational algebra and its extensions and functional query languages. Unification of type-expressions in this type system is an important issue since it allows one to perform type-checking and type-inference in a language independent way.

The general unification problem of type-expressions is shown to be intractable. A complete unification algorithm with a worst-time complexity in EXPTIME is presented. We also characterize a tractable sub-class of the unification for which the algorithm terminates in PTIME.

We are interested in applications of our results, particularly our tractable sub-class, to type-checking. By asking a user to partially instantiate a polymorphic function, we believe we can provide tractable type-checking. Also, we would like to extend the expressiveness of the type system to include union and recursive types, so one can express query languages for semi-structured data.

References

1. S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addision-Wesley Publishing Co., 1995.
2. P. Buneman and R. E. Frankel. FQL – a functional query language. In *SIGMOD*, pages 52–58, 1979.
3. P. Buneman and A. Ohori. Polymorphism and type inference in database programming. *ACM Transactions on Database Systems*, 21(1):30–76, March 1996.
4. L. Cardelli and J. C. Mitchell. Operations on records. *Mathematical Structures in Computer Science*, 1:3–48, 1991.
5. B. Carpenter. *The Logic of Typed Feature Structures*. Cambridge University Press, 1992.
6. L. Damas and R. Milner. Principal type schemes for functional programming. In *9th Symposium on Principles of Programming Languages*, 1982.
7. J. V. den Bussche and E. Waller. Polymorphic type inference for relational algebra. In *PODS*, pages 80–90, 1999.
8. M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness*. Freeman, 1979.
9. R. Kasper and W. C. Rounds. A logical semantics for feature structures. In *Proceedings of the 24th Annual Meeting of the Association for Computational Linguistics*, pages 257–266. Association for Computational Linguistics, 1986.
10. R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17, 1978.
11. A. Mycroft. Polymorphic type schemes and recursive definitions. In *6th International Conference on Programming*, 1984.
12. M. S. Paterson and M. N. Wegman. Linear unification. *Journal of Computer and System Sciences*, 16, 1978.
13. D. Rémy. Type inference for records in a natural extension of ML. In *TACS*, volume 789 of *Lecture Notes in Computer Science*. Springer, 1994.
14. S. Vansummeren. On the complexity of deciding typability in the relational algebra. *Acta Inf.*, 41(6):367–381, 2005.
15. M. Wand. Complete type inference for simple objects. In *LICS*, pages 37–44, 1987.