Polymorphic Type Inference for the Named Nested Relational Calculus

Jan Van den Bussche and Stijn Vansummeren*

Abstract

The named nested relational calculus is the canonical query language for the complex object database model and is equipped with a natural static type system. Given an expression in the language, without type declarations for the input variables, there is the problem of whether there are any input type declarations under which the expression is well-typed. Moreover, if there are, then which are they, and what is the corresponding output type for each of these? This problem is solved by a logic-based approach, and the decision problem is shown to be NP-complete.

1 Introduction

The named nested relational calculus (NNRC for short) is the canonical query language for the nested relational or complex object data model [1, 9, 34]. It is the natural extension to nested relations of the relational algebra and calculus, which form the basis of all contemporary database query languages [1].

Expressions in the NNRC are not always defined on every input. For example, the semantics of the expression $x.A$, which inspects the A attribute of variable x, is only well-defined if x holds a record with an attribute A. For this reason, the NNRC comes equipped with a static type system, which ensures type safety in the sense that every expression which passes the type system's tests is guaranteed to be well-defined.

The basic operators of the NNRC are polymorphic. We can inspect the A attribute of any record, as long as it has an attribute A. We can take the cartesian product of any two records whose attribute sets are disjoint. We can take the union of any two sets of the same type. Similar typing conditions can be formulated for the other operators of the NNRC. When combining operators into expressions, these typing conditions become more

*Address: Hasselt University, Department WNI, Agoralaan, gebouw D, B-3590 Diepenbeek, Belgium. Email: stijn.vansummeren@uhasselt.be

evolved. For example, for the expression

$$\{(x \times y).A \mid x \in R\}$$

to be well-typed, R must have a set type containing the type of x; x and y must have record types whose attribute sets are disjoint; and one of these attribute sets must contain A.

A natural question thus arises: given an NNRC expression e, under which assignments of free variables in e to types is e well-typed? And what is the resulting output type of e under these assignments? In particular, can we give an explicit description of the typically infinite collection of these *typings*? This is nothing but the NNRC version of the classical *type inference* problem. Type inference is an extensively studied topic in the theory of programming languages [21, 25], and is used in industrial-strength functional programming languages such as Standard ML [29] and Haskell [15].

Some expressions, for example $\emptyset.A$, are inherently untypable (i.e., these expressions do not admit any typing). Checking typability of an NNRC expression is the analog in NNRC of *type-checking* in implicitly typed programming languages with polymorphic type systems, such as ML. It is therefore interesting to see if typability is a decidable problem for the NNRC. If so, what is its complexity? It is already known for instance that typability for the particular case of the relational algebra is NP-complete [31, 32]. Also, it is P-complete for the simply typed lambda calculus [11] and EXPTIME-complete for ML [16, 17].

In this paper, we propose an explicit description of the set of all possible typings of an NNRC expression e by means of a conjunctive logical formula φ_e, which is interpreted in the universe of all possible types. The formula φ_e is efficiently computable from e. We proceed to show that the satisfiability problem of such conjunctive formulas belongs to NP. Consequently, typability for the NNRC is also in NP. Since the NNRC is an extension of the relational algebra, for which typability is already NP-complete, this thus shows that typability for the NNRC is not more difficult than for the special case of the relational algebra.

In the theory of programming languages one also finds type inference and type-checking algorithms for languages with sets and records, often in the presence of even more powerful features such as higher order functions [10, 23, 26, 27, 28, 33]. Indeed, the polymorphic type system of the NNRC can be encoded in the very general type inference framework of HM(X) [27, 28]. To our knowledge, however, we are the first to study the complexity of the typability problem for the specific type system of the NNRC.

Our motivations for this work are largely the same as for the previous work by one of us and Waller on the relational algebra [31]. We repeat some of these here. The main motivation is 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 (i.e., which types) it works, so that the programmer sees to which schema changes the query is sensitive.

Some features of modern database systems seem to add weight to the above motivation. *Stored procedures* [20] are 4GL and SQL code fragments stored in database dictionary tables. Whenever the schema changes, some of the stored procedures may become ill-typed, while others that were ill-typed may become well-typed. Having an explicit logical description of all typings of each stored procedure may be helpful in this regard. Models of *semi-structured* data [7, 14] 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. [8], 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 schema a given query is suitable.

A second motivation for this work stems from the area of database programming languages. A database programming language is a general-purpose programming language featuring an integrated database query language. The NNRC is, by design [6], a natural candidate for integration in a functional programming language such as the simply typed lambda calculus or ML. It is then an interesting question how this integration would affect the complexity of type-checking. Our results imply for example that adding the NNRC to the simply typed lambda calculus changes the complexity from P-complete to at least NP-hard. We further discuss this issue in Section 5.

A final goal of our paper is to provide an elementary, self-contained presentation of polymorphic type inference for the NNRC, accessible for researchers in database query languages who may not be familiar with type theory. For other work on database query languages related to typing issues, see the references [2, 4, 5, 13].

Organization We introduce the named nested relational calculus and its static type system in Section 2. In Section 3 we show that the set of all typings of an expression can be described by a logical formula. We show in Section 4 that satisfiability of such formulas is in NP, and provide concluding

discussions in Section 5.

2 Named Nested Relational Calculus

In this section we introduce the named nested relational calculus: its data model, its syntax, its semantics, and its type system. We start with the set-theoretic background used throughout this paper.

2.1 Set-theoretic background

We assume the reader to be familiar with the notions of union, intersection, difference, and cartesian product of sets (denoted by $X \cup Y$, $X \cap Y$, $X \setminus Y$, and $X \times Y$ respectively). We recall that two sets are considered to be equal if they contain precisely the same elements, and that a set X is a subset of a set Y ($X \subseteq Y$) if every element of X is also in Y.

We also recall that a mapping f from a set X to a set Y is a subset of $X \times Y$ such that for every $x \in X$ there exists exactly one $y \in Y$ with $(x, y) \in f$. In the following, we write $f\colon X \to Y$ to denote that f is a mapping from X to Y and we write $f(x)$ for the unique y assigned to x by f. Finally, recall that the restriction of f to a set $X' \subseteq X$ is the mapping from X' to Y that equals f on all $x \in X'$ (i.e., it is the mapping $f \cap X' \times Y$).

Note that, since mappings are sets, all notions about sets (such as union, intersection, ...) apply to mappings as well.

2.2 Data Model

We assume given a sufficiently large set $\{A, B, \dots\}$ of *attribute names*. A *row over* a set S is a mapping r from a finite set $dom(r)$ of attribute names to S. So, a row is a finite set of pairs. We write $\hat{\pi}_A(r)$ for the restriction of r to $dom(r) \setminus \{A\}$. We use an intuitive notation for rows, which we illustrate with an example. If r is the row with domain $\{A, B, C\}$ and $r(A) = a$, $r(B) = b$, and $r(C) = c$, then we write r as $\{A\colon a, B\colon b, C\colon c\}$.

We also assume given a recursively enumerable set $\mathbf{A} = \{a, b, \dots\}$ of *atoms*, which in practice will contain the usual data values such as integers, strings, and so on. A *value* v is either an atom, a record $[r]$ with r a row over values, or a finite set of values. We will denote values by v and w, rows over values by r and s, and finite sets of values by V and W. The natural join $[r] \bowtie [s]$ of two records is defined as follows:

$$[r] \bowtie [s] := \begin{cases} \{[r \cup s]\} & \text{if } r(A) = s(A) \text{ for all } A \in dom(r) \cap dom(s) \\ \emptyset & \text{otherwise.} \end{cases}$$

Note that, since r and s agree on their common attributes, $r \cup s$ is again a mapping; hence $r \cup s$ is again a row and $[r \cup s]$ is indeed a record.

2.3 Syntax

We assume given a sufficiently large set $\mathbf{X} = \{x, y, \dots\}$ of *variables*. The *named nested relational calculus* (NNRC for short) is the set of all expressions generated by the following grammar:

$$\begin{array}{rcl} e & ::= & x \\ & | & [] \mid [A\colon e] \mid e.A \mid e \times e \mid e \bowtie e \mid \hat{\pi}_A(e) \\ & | & \emptyset \mid \{e\} \mid e \cup e \mid \bigcup e \mid \{e \mid x \in e\} \\ & | & e = e \ ?\ e\ :\ e \end{array}$$

Here, e ranges over expressions, x ranges over variables, and A ranges over attribute names. We view expressions as abstract syntax trees and omit parentheses. The set $FV(e)$ of *free variables* of an expression e is defined as usual. That is, $FV(x) := \{x\}$, $FV(\emptyset) := \emptyset$, $FV(\{e_2 \mid x \in e_1\}) := FV(e_1) \cup (FV(e_2) \setminus \{x\})$, and $FV(e)$ is the union of the free variables of e's immediate subexpressions otherwise.

2.4 Semantics

Given values for its free variables, an expression evaluates to a new value. That is, expressions denote partial mappings from contexts to values, where a context is defined as follows.

Definition 1 (Context). A *context* σ is a mapping from a finite set of variables $dom(\sigma)$ to values. If e is an expression and $dom(\sigma)$ is a superset of $FV(e)$, then we say that σ is a *context on* e. We will denote by $x\colon v, \sigma$ the context σ' with domain $dom(\sigma) \cup \{x\}$ such that $\sigma'(x) = v$ and $\sigma'(y) = \sigma(y)$ for $y \neq x$.

The semantics of NNRC expressions is formally described by means of the *evaluation relation*, as defined in Figure 1. Here, we write $\sigma \models e \Rightarrow v$ to denote that e evaluates to value v under context σ on e. In the rule for $e_1 \times e_2$, note that $dom(r_1)$ and $dom(r_2)$ are required to be disjoint. This implies that $r_1 \cup r_2$ is again a mapping and that $[r_1 \cup r_2]$ is a record. It is easy to see that the evaluation relation is functional: an expression evaluates to at most one value under a given context. The evaluation relation is not total however. For example, if $\sigma(x)$ is an atom, then $x.A$ does not evaluate to any value under σ, since we can only inspect the attributes of records. Likewise, we can only concatenate disjoint records, join records, project out attributes of records, take the union of sets, flatten a set of sets, and iterate over sets. We will write $e(\sigma)$ for the unique value v for which $\sigma \models e \Rightarrow v$. If no such value exists, then we say that $e(\sigma)$ is *undefined*.

We note that the semantics of an expression only depends on its free variables: if two contexts σ and σ' on e are equal on $FV(e)$, then $\sigma \models e \Rightarrow v$ if, and only if, $\sigma' \models e \Rightarrow v$.

Variables

$$\frac{}{\sigma \models x \Rightarrow \sigma(x)}$$

Record operations

$$\frac{}{\sigma \models [\,] \Rightarrow [\emptyset]} \qquad \frac{\sigma \models e \Rightarrow v \quad r = \{A\colon v\}}{\sigma \models [A\colon e] \Rightarrow [r]} \qquad \frac{\sigma \models e \Rightarrow [r] \quad A \in dom(r)}{\sigma \models e.A \Rightarrow r(A)}$$

$$\frac{\sigma \models e_1 \Rightarrow [r_1] \quad \sigma \models e_2 \Rightarrow [r_2] \quad dom(r_1) \cap dom(r_2) = \emptyset}{\sigma \models e_1 \times e_2 \Rightarrow [r_1 \cup r_2]} \qquad \frac{\sigma \models e_1 \Rightarrow [r_1] \quad \sigma \models e_2 \Rightarrow [r_2]}{\sigma \models e_1 \bowtie e_2 \Rightarrow [r_1] \bowtie [r_2]}$$

$$\frac{\sigma \models e \Rightarrow [r] \quad A \in dom(r)}{\sigma \models \hat{\pi}_A(e) \Rightarrow [\hat{\pi}_A(r)]}$$

Set operations

$$\frac{}{\sigma \models \emptyset \Rightarrow \emptyset} \qquad \frac{\sigma \models e \Rightarrow v}{\sigma \models \{e\} \Rightarrow \{v\}} \qquad \frac{\sigma \models e_1 \Rightarrow V_1 \quad \sigma \models e_2 \Rightarrow V_2}{\sigma \models e_1 \cup e_2 \Rightarrow V_1 \cup V_2}$$

$$\frac{\sigma \models e \Rightarrow \{V_1, \ldots, V_n\}}{\sigma \models \bigcup e \Rightarrow V_1 \cup \cdots \cup V_n} \qquad \frac{\sigma \models e_1 \Rightarrow V \quad \forall v \in V : (x\colon v, \sigma) \models e_2 \Rightarrow w_v}{\sigma \models \{e_2 \mid x \in e_1\} \Rightarrow \{w_v \mid v \in V\}}$$

Conditional test

$$\frac{\sigma \models e_1 \Rightarrow v_1 \quad \sigma \models e_2 \Rightarrow v_2 \quad \sigma \models e_3 \Rightarrow v \quad v_1 = v_2}{\sigma \models e_1 = e_2 \;?\; e_3 \;:\; e_4 \Rightarrow v} \qquad \frac{\sigma \models e_1 \Rightarrow v_1 \quad \sigma \models e_2 \Rightarrow v_2 \quad \sigma \models e_4 \Rightarrow v \quad v_1 \neq v_2}{\sigma \models e_1 = e_2 \;?\; e_3 \;:\; e_4 \Rightarrow v}$$

Figure 1: The evaluation relation for NNRC expressions.

Example 1. Let *friends* and *John* be two variables. Suppose that the value of *friends* is a set of pairs of friends, as a set of records of the form $[\{Name\colon a, Friend\colon b\}]$ where *Name* and *Friend* are attributes. Suppose also that the value of *John* is a name (an atom). The following expression computes the set of all of *John*'s friends:

$$\bigcup\{x.Name = John\ ?\ \{x.Friend\}\ :\ \emptyset \mid x \in friends\}.$$

□

Note 1. Although we have not included the analog of the relational algebra renaming operation $\rho_{A/B}$, which renames the attribute A of a record to the attribute B, such an operation is expressible in the NNRC. Indeed, $\rho_{A/B}(x)$ can be expressed as $\hat{\pi}_A(x) \times [B\colon x.A]$. □

2.5 Type System

In order to ensure that an expression evaluates to a value for every input context in a desired set of contexts, the NNRC comes equipped with a *static type system*, which is defined as follows.

Definition 2 (Types). We assume given a finite set of *base types*. A *type* is a finite mathematical object, inductively defined as follows:

- every base type is a type;
- if ρ is a row over types, then $\mathbf{Record}(\rho)$ is a type; and
- if τ is a type, then $\mathbf{Set}(\tau)$ is a type.

(Recall that the notion of "row", used in the second item above, was introduced in Section 2.2). Two types τ and τ' are *equal* if they are the same mathematical object, i.e.,

- if τ and τ' are the same base type; or
- if $\tau = \mathbf{Record}(\rho)$ and $\tau' = \mathbf{Record}(\rho')$ with $dom(\rho) = dom(\rho')$ and $\rho(A)$ equal to $\rho'(A)$ for every $A \in dom(\rho)$; or
- if $\tau = \mathbf{Set}(\tau_1)$ and $\tau' = \mathbf{Set}(\tau_1')$ with τ_1 equal to τ_1'.

Definition 3 (Denotation of types). Every type τ *denotes* a set of values $[\![\tau]\!]$, which is inductively defined as follows:

- for each base type, $[\![\tau]\!]$ is a set of atoms, which we assume given;
- for types of the form $\mathbf{Record}(\rho)$, $[\![\mathbf{Record}(\rho)]\!]$ is the set of all records $[r]$ with $dom(r) = dom(\rho)$ and $r(A) \in [\![\rho(A)]\!]$, for every $A \in dom(r)$; and

- for types of the form $\mathbf{Set}(\tau)$, $[\![\mathbf{Set}(\tau)]\!]$ is the set of all finite sets over $[\![\tau]\!]$.

Definition 4 (Type assignment). A *type assignment* Γ is a mapping from a finite set $dom(\Gamma)$ of variables to types. We denote by $x\colon \tau, \Gamma$ the type assignment Γ' with domain $dom(\Gamma) \cup \{x\}$ such that $\Gamma'(x) = \tau$ and $\Gamma'(y) = \Gamma(y)$ for $y \neq x$. We extend $[\![\cdot]\!]$ to type assignments in the canonical way: $[\![\Gamma]\!]$ is the set of all contexts σ such that $dom(\sigma) = dom(\Gamma)$ and $\sigma(x) \in [\![\Gamma(x)]\!]$, for all $x \in dom(\Gamma)$. Finally, if $dom(\Gamma) \supseteq FV(e)$, then we say that Γ is a type assignment on e.

The *typing relation* for the NNRC is defined in Figure 2. Here we write $\Gamma \vdash e : \tau$ to indicate that expression e has type τ under type assignment Γ on e. Note that e has at most one type under Γ, which can easily be derived from Γ by applying the rules in an order determined by the syntax of expression e. If $\Gamma \vdash e : \tau$, then we call (Γ, τ) a *typing* of e.

We note that the type system is sound:

Proposition 1 (Soundness). *Let e be an expression, let Γ be a type assignment on e, and let τ be a type. If $\Gamma \vdash e : \tau$, then $e(\sigma)$ is defined and $e(\sigma) \in [\![\tau]\!]$, for every $\sigma \in [\![\Gamma]\!]$.*

The proof is by an easy induction on e. The type system is not "complete" however: there are examples of e, Γ, and τ such that $e(\sigma) \in [\![\tau]\!]$ for every $\sigma \in [\![\Gamma]\!]$, but yet $\Gamma \not\vdash e : \tau$. A simple example is the expression $e_0 = \emptyset \ ?\ [\,] \ :\ \emptyset$ where e_0 is an expression of set type that is actually unsatisfiable (i.e., it returns the empty set on every input). Since satisfiability of NNRC expressions is well-known to be undecidable, the above example actually shows that the following problem is undecidable:

Input:	e, Γ, τ
Decide:	Is $e(\sigma) \in \tau$ for every $\sigma \in \Gamma$.

Consequently, a sound and complete type system for the NNRC does not exist. In another paper [30] we have studied fragments of the nested relational calculus where such type systems do exist. In the current paper, we continue with the full language and the present type system which, though necessarily incomplete, is still very natural.

3 Type Inference

In this section we show that we can describe the set of all typings of an expression e by a logical formula. To this end, we first recall the definition of many-sorted first-order logic [12].

Variables

$$\frac{}{\Gamma \vdash x : \Gamma(x)}$$

Record operations

$$\frac{}{\Gamma \vdash [\,] : \mathbf{Record}(\emptyset)} \qquad \frac{\Gamma \vdash e : \tau \quad \rho = \{A\colon \tau\}}{\Gamma \vdash [A\colon e] : \mathbf{Record}(\rho)}$$

$$\frac{\Gamma \vdash e_1 : \mathbf{Record}(\rho_1) \quad \Gamma \vdash e_2 : \mathbf{Record}(\rho_2) \quad dom(\rho_1) \cap dom(\rho_2) = \emptyset}{\Gamma \vdash e_1 \times e_2 : \mathbf{Record}(\rho_1 \cup \rho_2)}$$

$$\frac{\Gamma \vdash e_1 : \mathbf{Record}(\rho_1) \quad \Gamma \vdash e_2 : \mathbf{Record}(\rho_2) \quad \rho_1(A) = \rho_2(A) \text{ for all } A \in dom(\rho_1) \cap dom(\rho_2)}{\Gamma \vdash e_1 \bowtie e_2 : \mathbf{Set}(\mathbf{Record}(\rho_1 \cup \rho_2))}$$

$$\frac{\Gamma \vdash e : \mathbf{Record}(\rho) \quad A \in dom(\rho)}{\Gamma \vdash e.A : \rho(A)} \qquad \frac{\Gamma \vdash e : \mathbf{Record}(\rho) \quad A \in dom(\rho)}{\Gamma \vdash \hat{\pi}_A(e) : \mathbf{Record}(\hat{\pi}_A(\rho))}$$

Set operations

$$\frac{\tau \text{ a type}}{\Gamma \vdash \emptyset : \mathbf{Set}(\tau)} \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \{e\} : \mathbf{Set}(\tau)} \qquad \frac{\Gamma \vdash e_1 : \mathbf{Set}(\tau) \quad \Gamma \vdash e_2 : \mathbf{Set}(\tau)}{\Gamma \vdash e_1 \cup e_2 : \mathbf{Set}(\tau)}$$

$$\frac{\Gamma \vdash e : \mathbf{Set}(\mathbf{Set}(\tau))}{\Gamma \vdash \bigcup e : \mathbf{Set}(\tau)} \qquad \frac{\Gamma \vdash e_1 : \mathbf{Set}(\tau_1) \quad x\colon \tau_1, \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \{e_2 \mid x \in e_1\} : \mathbf{Set}(\tau_2)}$$

Conditional test

$$\frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau' \quad \Gamma \vdash e_4 : \tau'}{\Gamma \vdash e_1 = e_2 \ ?\ e_3 \ :\ e_4 : \tau'}$$

Figure 2: The typing relation for NNRC expressions.

3.1 Many-Sorted First-Order Logic

Signatures, terms, and formulas. A *signature* Σ over a set of *sorts* S is a set consisting of (a possibly infinite number of) constant symbols, relation symbols, and function symbols. Every constant symbol c has an associated sort in S. Every relation symbol has an associated *arity* $\varsigma_1 \times \cdots \times \varsigma_n$, where every ς_i is a sort in S and $n > 0$. Likewise, every function symbol has an associated arity $\varsigma_1 \times \cdots \times \varsigma_n \to \varsigma_0$, where every ς_i is a sort in S and $n > 0$. We write $c\colon \varsigma$ to denote that c is a constant symbol of sort ς. We write $R\colon \varsigma_1 \times \cdots \times \varsigma_n \in \Sigma$ to denote that R has arity $\varsigma_1 \times \cdots \times \varsigma_n$. We use a similar notation for function symbols.

For every sort $\varsigma \in S$ we assume given an infinite collection of *variable symbols of sort* ς. Σ*-terms* are built from variable symbols, the constant symbols in Σ, and the function symbols in Σ as follows: every variable symbol x of sort ς is a Σ-term of sort ς, every constant symbol $c\colon \varsigma$ in Σ is a Σ-term of sort ς, and if $f\colon \varsigma_1 \times \cdots \times \varsigma_n \to \varsigma_0$ is a function symbol in Σ and $t_1, \ldots, t_n$ are Σ-terms of sort $\varsigma_1, \ldots, \varsigma_n$ respectively, then $f(t_1, \ldots, t_n)$ is a Σ-term of sort ς_0. *Atomic* Σ*-formulas* are formulas of the form $R(t_1, \ldots, t_n)$ where $R\colon \varsigma_1 \times \cdots \times \varsigma_n$ is a relation symbol in Σ and $t_1, \ldots, t_n$ are Σ-terms of sorts $\varsigma_1, \ldots, \varsigma_n$ respectively. *First-order* Σ*-formulas* are built up as usual from the atomic Σ-formulas and the logical connectives $\wedge$, $\neg$, and the existential quantifier $\exists$. We write $FO(\Sigma)$ for the set of all first-order Σ-formulas. With $FV(\varphi)$ we denote the set of all variables that occur free (i.e., not in the scope of some quantifier) in φ. Sometimes we write $\varphi(x_1, \ldots, x_n)$ to indicate that $FV(\varphi) \subseteq \{x_1, \ldots, x_n\}$. We say that φ is *quantifier free* if there is no quantifier in φ (i.e., if φ is a Boolean combination of atomic Σ-formulas).

Structures, valuations, and satisfaction. A Σ*-structure* $\mathcal{A}$ is a mapping assigning to every sort $\varsigma \in S$ a set $\mathcal{A}(\varsigma)$; to every constant symbol $c\colon \varsigma \in \Sigma$ an element $\mathcal{A}(c) \in \mathcal{A}(\varsigma)$; to every relation symbol $R\colon \varsigma_1 \times \cdots \times \varsigma_n \in \Sigma$ a set $\mathcal{A}(R) \subseteq \mathcal{A}(\varsigma_1) \times \cdots \times \mathcal{A}(\varsigma_n)$; and to every function symbol $f\colon \varsigma_1 \times \cdots \times \varsigma_n \to \varsigma_0 \in \Sigma$ a mapping $\mathcal{A}(f) : \mathcal{A}(\varsigma_1) \times \cdots \times \mathcal{A}(\varsigma_n) \to \mathcal{A}(\varsigma_0)$.

An $\mathcal{A}$*-valuation* is a mapping h from a finite set of variable symbols $dom(h)$ to $\bigcup_{\varsigma \in S} \mathcal{A}(\varsigma)$. $\mathcal{A}$-Valuations are extended to Σ-terms in the canonical way: $h(c) = \mathcal{A}(c)$ and $h(f(t_1, \ldots, t_n)) := \mathcal{A}(f)(h(t_1), \ldots, h(t_n))$. We write $x\colon a, h$ for the $\mathcal{A}$-valuation h' with domain $dom(h) \cup \{x\}$ such that $h'(x) = a$ and $h'(y) = h(y)$ for $y \neq x$.

Let φ be a first-order Σ-formula and suppose that $FV(\varphi) \subseteq dom(h)$. We say that h *satisfies* φ *in* $\mathcal{A}$, denoted by $\mathcal{A} \models \varphi(h)$, when:

- if φ is an atomic Σ-formula $R(t_1, \ldots, t_n)$, then $(h(t_1), \ldots, h(t_n)) \in \mathcal{A}(R)$;
- if φ is of the form $\varphi_1 \wedge \varphi_2$, then $\mathcal{A} \models \varphi_1(h)$ and $\mathcal{A} \models \varphi_2(h)$;

- if φ is of the form $\neg\varphi_1$, then $\mathcal{A} \not\models \varphi_1(h)$; and
- if φ is of the form $(\exists x)\varphi_1$ with x a variable symbol of sort ς, then there exists some $a \in A(\varsigma)$ such that $A \models \varphi_1(x : a, h)$.

3.2 Type Formulas

We will describe the set of all typings of an expression e by means of formulas in $FO(\Sigma_\tau)$. Here, Σ_τ is defined as the signature over the sorts $\{\mathit{type}, \mathit{row}\}$ consisting of

- a constant symbol ε of sort *row*;
- a binary relation symbol $=$ of arity $\mathit{type} \times \mathit{type}$;
- a binary relation symbol $\subseteq$ of arity $\mathit{row} \times \mathit{row}$;
- a binary relation symbol $\#$ of arity $\mathit{row} \times \mathit{row}$;
- a unary function symbol *Set* of arity $\mathit{type} \to \mathit{type}$;
- a unary function symbol *Record* of arity $\mathit{row} \to \mathit{type}$;
- for every attribute A, a unary function symbol A of arity $\mathit{type} \to \mathit{row}$; and
- a binary function symbol , of arity $\mathit{row} \times \mathit{row} \to \mathit{row}$.

We will interpret formulas in $FO(\Sigma_\tau)$ in the many-sorted structure $\mathcal{T}$ where

- $\mathcal{T}(\mathit{type})$ is the set of all types;
- $\mathcal{T}(\mathit{row})$ is the set of all rows over types;
- $\mathcal{T}(\varepsilon)$ is the empty row;
- $\mathcal{T}(=)$ relates equal types (with equality between types as in Definition 2);
- $\mathcal{T}(\subseteq)$ relates ρ to ρ' if ρ (as a mapping, i.e., a set of pairs) is a subset of ρ';
- $\mathcal{T}(\#)$ relates ρ to ρ' if $\mathit{dom}(\rho)$ is disjoint with $\mathit{dom}(\rho')$;
- $\mathcal{T}(\mathit{Set})$ maps τ to $\mathbf{Set}(\tau)$;
- $\mathcal{T}(\mathit{Record})$ maps ρ to $\mathbf{Record}(\rho)$;
- $\mathcal{T}(A)$ maps τ to the singleton row $\{A\colon \tau\}$; and
- $\mathcal{T}(,)$ is the "assymetric" concatenation operation: it maps ρ and ρ' to the row that equals ρ on $\mathit{dom}(\rho)$ and ρ' on $\mathit{dom}(\rho') \setminus \mathit{dom}(\rho)$.

Definition 5 (Type formula). A *type formula* is a formula in $FO(\Sigma_\tau)$ built up from atomic formulas using only existential quantifiers and conjunction.

Convention 1. It will be convenient to use the same set $\mathbf{X}$ from the syntax of the NNRC as the set of variable symbols of sort *type* in $FO(\Sigma_\tau)$. Variable symbols of sort *row* in $FO(\Sigma_\tau)$ will be denoted using letters from the beginning of the Greek alphabet.

Example 2. The following is an example of a type formula.

$$\varphi(x,y) \equiv (\exists\alpha)(\exists\beta)x = \mathit{Record}(\alpha) \wedge y = \mathit{Record}(\beta) \\ \wedge\, \alpha \# \beta \wedge (\exists z)A(z) \subseteq \alpha, \beta.$$

Evaluated on the structure $\mathcal{T}$, φ defines the set of all pairs of record types $(x = \mathbf{Record}(\rho_1), y = \mathbf{Record}(\rho_2))$ such that $\mathit{dom}(\rho_1) \cap \mathit{dom}(\rho_2) = \emptyset$ and $A \in \mathit{dom}(\rho_1) \cup \mathit{dom}(\rho_2)$. □

Definition 6. A type formula φ is *principal* for an expression e if φ defines the set of all typings of e. That is:

- φ contains no free variable symbols of sort *row*;
- the free variable symbols of sort *type* in φ are the free variables of e, plus one additional variable z; and
- $\Gamma \vdash e : \tau$ if, and only if, $\mathcal{T} \models \varphi(z\colon \tau, \Gamma)$.[1]

Example 3. For a simple example, consider the expression $e_1 = x \cup y$. Then the following is a principal type formula for e_1:

$$(\exists u)x = \mathit{Set}(u) \wedge y = \mathit{Set}(u) \wedge z = \mathit{Set}(u).$$

For a more complicated example, consider the expression:

$$e_2 = \{\{[B\colon t.A]\} \cup \{r \times s\} \mid t \in x \bowtie y\}.$$

Then the following is a principal type formula for e_2:

$$(\exists\alpha)(\exists\beta)(\exists\mu)(\exists\nu)x = \mathit{Record}(\alpha) \wedge y = \mathit{Record}(\beta) \wedge (\exists\beta')\alpha \subseteq \beta, \beta' \\ \wedge\, (\exists\alpha')\beta \subseteq \alpha, \alpha' \wedge r = \mathit{Record}(\mu) \wedge s = \mathit{Record}(\nu) \wedge \mu \# \nu \\ \wedge (\exists q)A(q) \subseteq \alpha, \beta \wedge B(q) \subseteq \mu, \nu \wedge \mu, \nu \subseteq B(q) \wedge z = \mathit{Set}(\mathit{Set}(\mathit{Record}(B(q))))$$

□

[1] We remind the reader of our convention that the variables from the syntax of the NNRC are used as variable symbols of sort *type* in $FO(\Sigma_\tau)$. Hence, every type assignment is a $\mathcal{T}$-valuation.

Theorem 2. *Every expression e has a principal type formula φ_e, of size linear in the size of e, and computable from e in polynomial time.*

Proof. Let e be an expression and let $x_1, \ldots, x_n$ be the free variables of e. Let z be a variable different from $x_1, \ldots, x_n$. We construct the principal type formula $\varphi_e(z, x_1, \ldots, x_n)$ for e by induction on e.

- Case $e = x$. Note that, since x is the only free variable of e, a principal type formula for e must have exactly two free variable symbols: x and z. Since every typing of e is of the form $(\Gamma, \Gamma(x))$ and conversely every $(\Gamma, \Gamma(x))$ is a typing of e, it suffices to define $\varphi_e := (z = x)$.

- Case $e = [\,]$. Note that, since e does not have any free variables, a principal type formula for e must have only one free variable symbol: z. Since every typing of e is of the form $(\Gamma, \mathbf{Record}(\emptyset))$ and every $(\Gamma, \mathbf{Record}(\emptyset))$ is a typing of e, it suffices to define:

$$\varphi_e := z = (\mathit{Record}(\varepsilon)).$$

- Case $e = [A\colon e']$. Note that every typing of e is of the form $(\Gamma, \mathbf{Record}(\{A\colon \tau)\})$. Moreover, since $(\Gamma, \mathbf{Record}(\{A\colon \tau)\})$ is a typing of e if, and only if, (Γ, τ) is a typing of e', it suffices to define:

$$\varphi_e := (\exists x_0)\varphi_{e'}(x_0, x_1 \ldots, x_n) \wedge z = \mathit{Record}(A(x_0)).$$

- Case $e = e'.A$. Since (Γ, τ) is a typing of e if, and only if, $(\Gamma, \mathbf{Record}(\rho))$ is a typing of e' with $\rho(A) = \tau$, it suffices to define:

$$\varphi_e := (\exists x_0)\varphi_{e'}(x_0, x_1 \ldots, x_n) \wedge (\exists \alpha)x_0 = \mathit{Record}(A(z), \alpha).$$

- Case $e = e_1 \times e_2$. Let $y_1, \ldots, y_k$ be the free variables of e_1 and let $y'_1, \ldots, y'_l$ be the free variables of e_2. Note that every typing of e is of the form $(\Gamma, \mathbf{Record}(\rho))$. Since $(\Gamma, \mathbf{Record}(\rho))$ is a typing of e if, and only if, there exist rows ρ_1 and ρ_2 such that $(\Gamma, \mathbf{Record}(\rho_1))$ is a typing of e_1; $(\Gamma, \mathbf{Record}(\rho_2))$ is a typing of e_2; $\mathit{dom}(\rho_1)$ is disjoint with $\mathit{dom}(\rho_2)$; and $\rho = \rho_1 \cup \rho_2$, it suffices to define:

$$\begin{aligned}\varphi_e := (\exists y_0)\varphi_{e_1}(y_0, y_1 \ldots, y_k) \wedge (\exists \alpha)y_0 = \mathit{Record}(\alpha)\\ \wedge (\exists y'_0)\ \varphi_{e_2}(y'_0, y'_1, \ldots, y'_l) \wedge (\exists \alpha')y'_0 = \mathit{Record}(\alpha')\\ \wedge \alpha \,\#\, \alpha' \wedge z = \mathit{Record}(\alpha, \alpha').\end{aligned}$$

- Case $e = e_1 \bowtie e_2$. Let $y_1, \ldots, y_k$ be the free variables of e_1 and let $y'_1, \ldots, y'_l$ be the free variables of e_2. Note that every typing of e is of the form $(\Gamma, \mathbf{Set}(\mathbf{Record}(\rho)))$. Since $(\Gamma, \mathbf{Set}(\mathbf{Record}(\rho)))$ is a typing of e if, and only if, there exist rows ρ_1 and ρ_2 such that $(\Gamma, \mathbf{Record}(\rho_1))$

is a typing of e_1; $(\Gamma, \mathbf{Record}(\rho_2))$ is a typing of e_2; $\rho_1(A) = \rho_2(A)$ for every $A \in dom(\rho_1) \cap dom(\rho_2)$; and $\rho = \rho_1 \cup \rho_2$, it suffices to define:

$$\begin{aligned}\varphi_e := (\exists y_0)\varphi_{e_1}(y_0, y_1 \ldots, y_k) \wedge (\exists \alpha) y_0 = \mathit{Record}(\alpha) \\ \wedge (\exists y_0')\ \varphi_{e_2}(y_0', y_1', \ldots, y_l') \wedge (\exists \alpha') y_0' = \mathit{Record}(\alpha') \\ \wedge (\exists \beta')\alpha \subseteq \alpha', \beta' \wedge (\exists \beta)\alpha' \subseteq \alpha, \beta \\ \wedge z = \mathit{Set}(\mathit{Record}(\alpha, \alpha')).\end{aligned}$$

Indeed, the subformula $(\exists \beta')\alpha \subseteq \alpha', \beta' \wedge (\exists \beta)\alpha' \subseteq \alpha, \beta$ ensures that the rows held by α and β agree on the common attributes in their domain.

- Case $e = \hat{\pi}_A(e')$. Note that every typing of e is of the form $(\Gamma, \mathbf{Record}(\rho))$. Since $(\Gamma, \mathbf{Record}(\rho))$ is a typing of e if, and only if, there exists a row ρ' such that $(\Gamma, \mathbf{Record}(\rho'))$ is a typing of e' and $\rho = \hat{\pi}_A(\rho')$, it suffices to define:

$$\begin{aligned}\varphi_e := (\exists x_0)\varphi_{e'}(x_0, x_1 \ldots, x_n) \wedge (\exists \alpha) x_0 = \mathit{Record}(\alpha) \\ \wedge (\exists \beta)(\exists y) A(y) \# \beta \wedge \alpha \subseteq A(y), \beta \wedge A(y), \beta \subseteq \alpha \\ \wedge z = \mathit{Record}(\beta).\end{aligned}$$

Indeed, the subformula $A(y) \# \beta \wedge \alpha \subseteq A(y), \beta \wedge A(y), \beta \subseteq \alpha$ ensures that attribute A is not in the domain of the row held by β and that the row held by α equals the row held by β on all other attributes.

- Case $e = \emptyset$. Note that, since e does not have any free variables, a principal type formula for e must have only one free variable symbol: z. Since every typing of e is of the form $(\Gamma, \mathbf{Set}(\tau))$ and conversely every $(\Gamma, \mathbf{Set}(\tau))$ is a typing of e, it suffices to define $\varphi_e := (\exists y) z = \mathit{Set}(y)$.

- Case $e = \{e'\}$. Note that every typing of e is of the form $(\Gamma, \mathbf{Set}(\tau))$. Since $(\Gamma, \mathbf{Set}(\tau))$ is a typing of e if, and only if, (Γ, τ) is a typing of e', it suffices to define:

$$\varphi_e := (\exists x_0)\varphi_{e'}(x_0, x_1, \ldots, x_n) \wedge z = \mathit{Set}(x_0).$$

- Case $e = e_1 \cup e_2$. Let $y_1, \ldots, y_k$ be the free variables of e_1 and let $y_1', \ldots, y_l'$ be the free variables of e_2. Note that every typing of e is of the form $(\Gamma, \mathbf{Set}(\tau))$. Since $(\Gamma, \mathbf{Set}(\tau))$ is a typing of e if, and only if, $(\Gamma, \mathbf{Set}(\tau))$ is a typing of e_1 and $(\Gamma, \mathbf{Set}(\tau))$ is a typing of e_2, it suffices to define:

$$\begin{aligned}\varphi_e := (\exists y_0)\varphi_{e_1}(y_0, y_1 \ldots, y_k) \wedge (\exists y_0')\varphi_{e_2}(y_0', y_1', \ldots, y_l') \\ \wedge y_0 = y_0' \wedge z = y_0 \wedge (\exists x) z = \mathit{Set}(x).\end{aligned}$$

- Case $e = \bigcup e'$. Note that every typing of e is of the form $(\Gamma, \mathbf{Set}(\tau))$. Since $(\Gamma, \mathbf{Set}(\tau))$ is a typing of e if, and only if, $(\Gamma, \mathbf{Set}(\mathbf{Set}(\tau)))$ is a typing of e', it suffices to define:

$$\varphi_e := (\exists x_0)\varphi_{e'}(x_0, x_1, \ldots, x_n) \wedge (\exists y)x_0 = \mathit{Set}(\mathit{Set}(y)) \wedge z = \mathit{Set}(y).$$

- Case $e = \{e_2 \mid x \in e_1\}$. Let $y_1, \ldots, y_k$ be the free variables of e_1 and let $x, y'_1, \ldots, y'_l$ be the free variables of e_2. Note that every typing of e is of the form $(\Gamma, \mathbf{Set}(\tau))$. Since $(\Gamma, \mathbf{Set}(\tau))$ is a typing of e if, and only if, there exists a type τ' such that $(\Gamma, \mathbf{Set}(\tau'))$ is a typing of e_1 and $((x : \tau', \Gamma), \tau)$ is a typing of e_2, it suffices to define:

$$\begin{aligned}\varphi_e := (\exists y_0)\varphi_{e_1}(y_0, y_1, \ldots, y_k) \wedge (\exists x)y_0 = \mathit{Set}(x) \\ \wedge (\exists y'_0)\varphi_{e_2}(y'_0, x, y'_1, \ldots, y'_l) \wedge z = \mathit{Set}(y'_0).\end{aligned}$$

- Case $e = e_1 = e_2 \ ? \ e_3 \ : \ e_4$. Let $u_1, \ldots, u_k$ be the free variables of e_1, let $u'_1, \ldots, u'_l$ be the free variables of e_2, let $y_1, \ldots, y_p$ be the free variables of e_3, and let $y'_1, \ldots, y'_q$ be the free variables of e_4. Note that every typing of e is of the form (Γ, τ). Since (Γ, τ) is a typing of e if, and only if, there exists a type τ' such that (Γ, τ') is a typing of e_1 and e_2 and (Γ, τ) is a typing of e_3 and e_4, it suffices to define:

$$\begin{aligned}(\exists u_0)\varphi_{e_1}(u_0, u_1, \ldots, u_k) \wedge (\exists u'_0)\varphi_{e_2}(u'_0, u'_1, \ldots, u'_l) \wedge u_0 = u'_0 \\ \wedge (\exists y_0)\varphi_{e_3}(y_0, y_1, \ldots, y_p) \wedge (\exists y'_0)\varphi_{e_4}(y'_0, y'_1, \ldots, y'_q) \wedge y'_0 = y_0 \\ \wedge z = y_0.\end{aligned}$$

Clearly, φ_e is of size linear in the size of e, and is computable from e in polynomial time. □

4 Typability

Some expressions, such as for example $[\,].A$, do not have any typing. We will refer to such expressions as *untypable*.

Definition 7. An expression e is called *typable* if there exists a type assignment Γ on e and a type τ such that $\Gamma \vdash e : \tau$. Deciding whether a given expression e is typable is called the *typability problem*.

Example 4. Some additional examples of untypable formulas are $x \cup x.A$, $[A\colon x].B$, and $x.A \bowtie (x \times [A\colon y])$. □

It follows from Theorem 2 that deciding whether an expression e is typable is equivalent to computing the principal type formula φ_e for e and

then deciding whether φ_e is satisfiable in $\mathcal{T}$. We will now show that deciding the latter is in the complexity class NP. Since φ_e is computable from e in polynomial time, it then follows that the typability problem is also in NP.

We first note that, since φ_e is a conjunctive formula, it is very easily put in existential prenex normal form $(\exists x_1)\ldots(\exists x_n)\psi$ with ψ quantifier free. Clearly, φ_e is satisfiable in $\mathcal{T}$ if, and only if, ψ is. We will therefore restrict our attention to quantifier free type formulas.

Definition 8. The set *Specattrs*(φ) of a type formula φ is the set of attributes A for which a term of the form $A(t)$ occurs in φ.

Definition 9. The *deep restriction* $\rho|_S$ of a row over types ρ to a set of attributes S is the row ρ' with domain $dom(\rho) \cap S$ such that for each $A \in dom(\rho) \cap S$, $\rho'(A)$ is the deep restriction of the type $\rho(A)$ to S. Here, the deep restriction $\tau|_S$ of a type τ to S is the type obtained from τ by deep-restricting every row occurring in τ to S. So, this is a recursive definition. In addition, we define the deep restriction $h|_S$ of a $\mathcal{T}$-valuation h to S as the $\mathcal{T}$-valuation h' such that $h'(x) = h(x)|_S$ for every $x \in dom(h)$.

Lemma 3. *If φ is a type formula and h is a $\mathcal{T}$-valuation such that $\mathcal{T} \models \varphi(h)$, then also $\mathcal{T} \models \varphi(h|_{Specattrs(\varphi)})$.*

Proof. It is easy to see by induction on t that, for any term t in $FO(\Sigma_\tau)$ we have $h|_{Specattrs(\varphi)}(t) = h(t)|_{Specattrs(\varphi)}$. The lemma then follows by an easy induction on φ. □

Theorem 4. *Deciding satisfiability in $\mathcal{T}$ of a quantifier free type formula is in* NP.

Proof. Let ψ be a quantifier free type formula. Let, for every attribute name A and every variable α of sort *row* in ψ, x_A^α be a distinct type variable not in ψ. An *attribute assignment* on ψ is a mapping f that assigns to each variable α of sort *row* in ψ to a term in $FO(\Sigma_\tau)$ of sort *row* of the form

$$A(x_A^\alpha), \ldots, B(x_B^\alpha), \varepsilon$$

where $\{A, \ldots, B\} \subseteq Specattrs(\psi)$. Note that, in particular, the size of f is polynomial in the size of ψ. Let ψ_f be the quantifier-free type formula obtained from ψ by replacing each variable α of sort *row* in ψ by the term $f(\alpha)$. Clearly, ψ_f can be computed from e in polynomial time.

We now claim that ψ is satisfiable in $\mathcal{T}$ if, and only if, there exists an attribute assignment f on ψ such that ψ_f is satisfiable in $\mathcal{T}$. Indeed, suppose that ψ is satisfiable in $\mathcal{T}$. By Lemma 3 there exists a valuation h of ψ such that $\mathcal{T} \models \psi(h)$ and such that $dom(h(\alpha)) \subseteq Specattrs(\psi)$, for all variables α of sort *row* in ψ. Then let f be the attribute assignment on ψ defined by

$$f(\alpha) := A(x_A^\alpha), \ldots, B(x_B^\alpha), \varepsilon$$

where $dom(h(\alpha)) = \{A, \ldots, B\}$. Let h_f be the valuation on ψ_f which equals h on type variables in ψ and for which $h_f(x^\alpha_A) = h(\alpha)(A)$. It is easy to see that $\mathcal{T} \models \psi_f(h_f)$.

Conversely, suppose that there exists an attribute assignment f on ψ such that ψ_f is satisfiable in $\mathcal{T}$. Then let h_f be a valuation of ψ_f such that $\mathcal{T} \models \psi_f(h_f)$. Let h be the valuation on ψ which equals h_f on the type variables in ψ and for which $h(\alpha)$ is the row ρ with domain $\{A, \ldots, B\}$ where $f(\alpha) = A(x^\alpha_A), \ldots, B(x^\alpha_B), \varepsilon$ such that $\rho(A) = h_f(x^A_\alpha)$. It is easy to see that $\mathcal{T} \models \psi(h)$.

Hence, in order to check satisfiability of ψ, it suffices to guess an attribute assignment on ψ (which is polynomial in the size of ψ) and check whether ψ_f is satisfiable. The latter can be done in polynomial time, as we show in the following theorem. □

Theorem 5. *Satisfiability in $\mathcal{T}$ of quantifier free type formulas without variables of sort row can be decided in polynomial time.*

Proof. Let ψ be a quantifier free type formula without variables of sort *row*. Since ψ is a conjunction of atomic formulas, we can view ψ as a set of atomic formulas. Moreover, since there are no variables of sort *row* in ψ, we can treat every term t of sort *row* in ψ as a row over terms of sort type. For example, we can treat the term $t = A(x), \varepsilon, B(y), A(z), \varepsilon, C(u)$ as the row $\{A\colon x, B\colon y, C\colon u\}$. Likewise, we can treat the term $t' = \varepsilon$ as the empty row. Then let ψ_1 be the subset of ψ defined by

$$\psi_1 := \{u_1 \# u_2 \mid (u_1 \# u_2) \in \psi\} \cup \{u_1 \subseteq u_2 \mid (u_1 \subseteq u_2) \in \psi\}.$$

Let ψ_2 be defined by

$$\begin{aligned}\psi_2 := &\{t_1 = t_2 \mid (t_1 = t_2) \in \psi\} \\ &\cup \{u_1(A) = u_2(A) \mid (u_1 \subseteq u_2) \in \psi \text{ and } A \in dom(u_1) \cap dom(u_2)\}.\end{aligned}$$

It is clear that ψ_1 and ψ_2 can be computed from ψ in polynomial time. Let us call ψ_1 *consistent* if for every $u_1 \# u_2$ in ψ_1 we have $dom(u_1) \cap dom(u_2) = \emptyset$ and for every $u_1 \subseteq u_2$ we have $dom(u_1) \subseteq dom(u_2)$.

We claim that ψ is satisfiable in $\mathcal{T}$ if, and only if, ψ_1 is consistent and ψ_2 is satisfiable in $\mathcal{T}$. Indeed, it is easy to see that if ψ is satisfiable, then ψ_1 must be consistent. Furthermore, if h is a valuation for which $\mathcal{T} \models \psi(h)$, then $h(t_1) = h(t_2)$ for every $t_1 = t_2$ in ψ and $h(u_1)(A) = h(u_2)(A)$ for every $u_1 \subseteq u_2$ in ψ and every $A \in dom(u_1) \cap dom(u_2)$. Hence, $\mathcal{T} \models \psi_2(h)$.

Conversely, suppose that ψ_1 is consistent and that ψ_2 is satisfiable in $\mathcal{T}$. Let h be a valuation such that $\mathcal{T} \models \psi_2(h)$. Then $h(t_1) = h(t_2)$ for every $t_1 = t_2$ in ψ. Furthermore, since $dom(u_1) \cap dom(u_2) = \emptyset$ for every $u_1 \# u_2$ in ψ (as ψ_1 is consistent), and since there are no variables of sort *row* in ψ, it follows that $dom(h(u_1)) \cap dom(h(u_2)) = \emptyset$ for every $u_1 \# u_2$

in ψ. Finally, since $dom(u_1) \subseteq dom(u_2)$ for every $u_1 \subseteq u_2$ in ψ (as ψ_1 is consistent) and since $h(u_1)(A) = h(u_2)(A)$ for every $A \in dom(u_1) \cap dom(u_2)$ (as $\mathcal{T} \models \psi_2(h)$), it follows that $h(u_1) \subseteq h(u_2)$ for every $u_1 \subseteq u_2$ in ψ. Hence, $\mathcal{T} \models \psi(h)$.

In order to check satisfiability of ψ, it hence suffices to check consistency of ψ_1 and satisfiability of ψ_2. Consistency of ψ_1 can clearly be checked in polynomial time. We now show that satisfiability of ψ_2 in $\mathcal{T}$ can also be checked in polynomial time. Let $\prec$ be some arbitrarily fixed order on the special attributes of ψ_2. We assume without loss of generality that every term of sort *row* in ψ_2 is of the form $A_1(t_1), A_2(t_2), \ldots, A_m(t_m), \varepsilon$ with $A_1 \prec A_2 \prec \cdots \prec A_m$ (as such terms can clearly be rewritten into this form in polynomial time without affecting satisfiability otherwise). Note that ψ_2 is simply a set of equations between terms of sort type. It is then easy to see that checking satisfiability of ψ_2 in $\mathcal{T}$ amounts to finding a substitution θ of variables in ψ_2 to terms in $FO(\Sigma_\tau)$ of sort *type* such that $\theta(t_1)$ and $\theta(t_2)$ are syntactically equal for every equation $t_1 = t_2$ in ψ_2. Hence, satisfiability of ψ_2 reduces to finding a unifier of every equation in ψ_2, which is known to be decidable in polynomial time [3, 18, 24]. □

The complexity upper bound of NP provided by Theorem 4 is actually tight:

Proposition 6. *Typability for the NNRC is* NP*-complete.*

Proof. Since typability of an expression e is equivalent to computing the type formula φ_e for e and then deciding whether φ_e is satisfiable in $\mathcal{T}$, it follows from Theorems 2 and 4 that typability for the NNRC is in NP.

It is already known that typability for the relational algebra is NP-complete [31, 32]. It is also well-known that the relational algebra can be simulated in the NNRC [9, 34]. It is not difficult to see that this simulation preserves typability. Hence, typability for the NNRC is also NP-complete. □

By the reduction of typability of an NNRC expression to satisfiability in $\mathcal{T}$ of a type formulas it also follows:

Corollary 7. *Deciding satisfiability in $\mathcal{T}$ of a type formula is* NP*-complete.*

5 Concluding Remarks

Simplification of principal type formulas. We have shown that the set of all typings of an NNRC expression e can be explicitly described by a conjunctive formula φ_e in $FO(\Sigma_\tau)$, which is efficiently computable from e. From a practical viewpoint our definition of a principal type formula is deficient, however. Indeed, a principal type formula for a program is generally

expected to be a useful, concise, and easily understandable abstraction of what the program does. For example, if we view the well-known type inference algorithm for the programming language ML in our setting, a principal type formula is either either *false* (meaning the function whose type we are infering is untypable) or of the form

$$(\exists u_1)\dots(\exists u_m)\, z = t \wedge x_1 = t_1 \wedge \dots \wedge x_n = t_n.$$

Here, $u_1, \dots, u_m$ are variable symbols, and $t, t_1, \dots, t_n$ are terms of sort *type* built from $u_1, \dots, u_m$. For such formulas it is easy to discern the kinds of types that can be assigned to $z, x_1, \dots, x_n$. In contrast, we allow arbitrary complex type formulas. Consider, for example, the principal type formula for $[C\colon x \cup y]$ that is output by the inductive algorithm given in the proof of Theorem 2:

$$\begin{aligned}\varphi_1 \equiv (\exists u_0)(\exists u_1)u_1 = x \wedge (\exists u_1')u_1' = y \wedge u_1 = u_1' \wedge u_0 = u_1 \\ \wedge (\exists u_2)u_0 = \mathbf{Set}(u_2) \wedge z = \mathbf{Record}(C(u_2)).\end{aligned}$$

The extra use of bound variables and equality predicates makes this formula harder to understand than its equivalent φ_2 in "ML normal form":

$$\varphi_2 \equiv (\exists u_2)z = \mathbf{Record}(C(\mathbf{Set}(u_2))) \wedge x = \mathbf{Set}(u_2) \wedge y = \mathbf{Set}(u_2).$$

For presentation of principal type formulas to the programmer, we would hence like to have a normal form that allows formulas like φ_2, but avoids needlessly complex formulas like φ_1. Moreover, such a normal form should come with a simplification algorithm that puts arbitrary principal type formulas in this normal form.

The ML normal form given above does not suffice for this purpose, as not all type formulas can be expressed in it. For example, a principal type formula for $x \times y$ must express that the row of the record in x is disjoint with the row of the record in y. Therefore, such a type formula must contain an atomic formula of the form $t_1 \# t_2$, which cannot occur in a type formula in ML normal form. We could therefore generalize the ML normal form to

$$(\exists u_1)\dots(\exists u_m)\, z = t \wedge x_1 = t_1 \wedge \dots \wedge x_n = t_n \wedge \psi.$$

Here, $u_1, \dots, u_m$ are variable symbols; $t, t_1, \dots, t_n$ are terms of sort type built from $u_1, \dots, u_m$; and ψ is a quantifier free formula in $FO(\Sigma_\tau)$ that only contains atomic formulas of the form $t_1' \subseteq t_2'$ and $t_1' \# t_2'$ such that $FV(\psi) \subseteq \{u_1, \dots, u_m\}$. We call ψ the *constraint part*.

It is not difficult to show that every principal type formula has an equivalent formula in this form. Unfortunately, it is unsuitable for presentation purposes, as it still allows arbitrary complex type formulas. For example,

the following principal type formula for $[C\colon x \cup y]$ has the above form, but is as complex as φ_1:

$$\begin{aligned}\varphi_3 \equiv (\exists u_0)(\exists u_1)(\exists u_2)(\exists u_1')z = \mathbf{Record}(C(u_0)) \wedge x = u_1 \wedge y = u_1' \\ \wedge A(u_1) \subseteq A(u_1') \wedge A(u_0) \subseteq A(u_1) \wedge A(u_0) \subseteq A(\mathbf{Set}(u_2)).\end{aligned}$$

Indeed, here atomic formulas of the form $t_1 = t_2$ in φ_1 are simply replaced by atomic formulas $A(t_1) \subseteq A(t_2)$, resulting in the same obfuscation. To overcome a similar problem, Odersky, Sulzmann, and Wehr [22, 28] propose to further restrict the constraint part ψ to be *logically equation-free*. A formula is logically equation-free if only trivial equations are logically implied by it. That is, for any two terms t_1 and t_2, if $h(t_1) = h(t_2)$ for every valuation h such that $\mathcal{T} \models \psi(h)$, then t_1 should be syntactically equal to t_2. This restriction rules out φ_3 above, since the constraint part

$$A(u_1) \subseteq A(u_1') \wedge A(u_0) \subseteq A(u_1) \wedge A(u_0) \subseteq A(\mathbf{Set}(u_2))$$

logically implies the equations $u_1 = u_0$, $u_0 = \mathbf{Set}(u_2)$, and so on. We currently do not know, however, if every principal type formula has an equivalent formula in this restricted normal form, and, if so, if such an equivalent formula is effectively computable.

Note that, even after simplification, principal type formulas may be quite complex. For example, recall the expression e_2 from Example 3:

$$e_2 = \{\{[B\colon t.A]\} \cup \{r \times s\} \mid t \in x \bowtie y\}.$$

Then the following is a principal type formula for e_2:

$$\begin{aligned}\varphi_4 \equiv (\exists\alpha)(\exists\alpha')(\exists\beta)(\exists\beta')(\exists\mu)(\exists\nu)(\exists q)\ z = \mathit{Set}(\mathit{Set}(\mathit{Record}(B(q)))) \\ \wedge x = \mathit{Record}(\alpha) \wedge y = \mathit{Record}(\beta) \wedge r = \mathit{Record}(\mu) \wedge s = \mathit{Record}(\nu) \\ \wedge \alpha \subseteq \beta, \beta' \wedge \beta \subseteq \alpha, \alpha' \wedge \mu \# \nu \wedge A(q) \subseteq \alpha, \beta \\ \wedge B(q) \subseteq \mu, \nu \wedge \mu, \nu \subseteq B(q).\end{aligned}$$

Note that φ_4 is of the restricted form proposed by Odersky, Sulzmann, and Wehr, but is still complex. This complexity is entirely due to the complicated typing rules for $\times$, $\bowtie$, and $\hat{\pi}_A$. This is not solely a deficiency in our approach: other type systems treating such record operations [10, 22, 28] also suffer from this problem.

Typing database programming languages. As we have already mentioned in the Introduction, our results on the complexity of typability can be used to determine how the integration of the NNRC in an implicitly typed functional programming language, such as the simply typed lambda calculus or ML, affects the complexity of type-checking in this language. Adding the

NNRC to the simply typed lambda calculus for example changes the complexity from P-complete to at least NP-hard. The type-checking problem for ML is known to be EXPTIME-complete [16, 17]. Hence, our result that type-checking the NNRC is NP-hard does not necessarily imply that type-checking the integrated ML-NNRC is harder than type-checking ML. We should note, however, that the EXPTIME-completeness for ML arises only due to programs of a very particular form, which rarely occur in practice [19]. The ML type-checking algorithms therefore typically run in linear time in practice [19]. The NP-hardness of type-checking in the NNRC on the other hand arises in many expressions, due to many different reasons [32]. It is therefore likely that type-checking the integrated ML-NNRC language will in practice be slower than type-checking ML.

References

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations Of Databases*. Addison-Wesley, 1995.

[2] Suad Alagic. Type-checking OQL queries in the ODMG type systems. *ACM Transactions on Database Systems*, 24(3):319–360, 1999.

[3] Franz Baader and Wayne Snyder. Unification theory. In *Handbook of Automated Reasoning*, pages 445–532. Elsevier and MIT Press, 2001.

[4] Herman Balsters, Rolf A. de By, and Roberto Zicari. Typed sets as a basis for object-oriented database schemas. In *ECOOP'93 - Object-Oriented Programming*, volume 707 of *Lecture Notes in Computer Science*, pages 161–184. Springer, 1993.

[5] Catriel Beeri and Tova Milo. Subtyping in OODBs. *Journal of Computer and System Sciences*, 51(2):223–243, 1995.

[6] Val Breazu-Tannen, Peter Buneman, and Limsoon Wong. Naturally embedded query languages. In *Proceedings of the 4th International Conference on Database Theory, Berlin, Germany, October 1992*, volume 646 of *Lecture Notes in Computer Science*, pages 140–154. Springer-Verlag, 1992.

[7] Peter Buneman, Susan Davidson, Gerd Hillebrand, and Dan Suciu. A query language and optimization techniques for unstructured data. *SIGMOD Record*, 25(2):505–516, 1996.

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

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

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

[11] Cynthia Dwork, Paris C. Kanellakis, and John C. Mitchell. On the sequential nature of unification. *Journal of Logic Programming*, 1(1):35–50, 1984.

[12] Herbert B. Enderton. *A Mathematical Introduction to Logic*. Academic Press, 2001. Second Edition.

[13] Mary F. Fernández, Jérôme Siméon, and Philip Wadler. A semi-monad for semi-structured data. In *Database Theory - ICDT 2001*, volume 1973 of *Lecture Notes in Computer Science*, pages 263–300. Springer, 2001.

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

[15] Simon Peyton Jones. *Haskell 98 Language and Libraries*. Cambridge University Press, 2003.

[16] Paris C. Kanellakis, Harry G. Mairson, and John C. Mitchell. Unification and ML-type reconstruction. In *Computational Logic - Essays in Honor of Alan Robinson*, pages 444–478. MIT Press, 1991.

[17] Harry G. Mairson. Deciding ML typability is complete for deterministic exponential time. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, 1990.

[18] Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4(2), 1982.

[19] David A. McAllester. Joint rta-tlca invited talk: A logical algorithm for ml type inference. In *Rewriting Techniques and Applications, 14th International Conference, RTA 2003, Valencia, Spain, June 9-11, 2003, Proceedings*, volume 2706 of *Lecture Notes in Computer Science*, pages 436–451. Springer, 2003.

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

[21] John C. Mitchell. *Foundations for Programming Languages.* MIT Press, 1996.

[22] Martin Odersky, Martin Sulzmann, and Martin Wehr. Type inference with constrained types. *Theory and Practice of Object Systems*, 5(1):35–55, 1999.

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

[24] Mike S. Paterson and Mark N. Wegman. Linear unification. *Journal of Computer and System Sciences*, 16:158–167, 1978.

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

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

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

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

[29] Jeffrey D. Ullman. *Elements of ML Programming, ML97 Edition.* Prentice-Hall, 1998.

[30] Jan Van den Bussche, Dirk Van Gucht, and Stijn Vansummeren. Well-definedness and semantic type-checking in the nested relational calculus and xquery. In *Database Theory - ICDT 2005*, volume 3363 of *Lecture Notes in Computer Science*, pages 99–113. Springer, 2005.

[31] Jan Van den Bussche and Emanuel Waller. Polymorphic type inference for the relational algebra. *Jounal of Computer and System Sciences*, 64:694–718, 2002.

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

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

[34] Limsoon Wong. *Querying nested collections*. PhD thesis, University of Pennsylvania, 1994.