# The Complexity of Higher-order Queries

Michael Benedikt[a], Gabriele Puppis[b], Huy Vu[c]

[a] *Department of Computer Science, Oxford University, Parks Rd, Oxford OX1 3QD, UK*
[b] *CNRS / LaBRI, 351 Cours de la Libration, Talence 33405, France*
[c] *Science Park, Oxford OX4 4GA, UK*

**Abstract**

Higher-order transformations are ubiquitous within data management. In relational databases, higher-order queries appear in numerous aspects including query rewriting and query specification. This work investigates languages that combine higher-order transformations with ordinary relational database query languages. We study the two most basic computational problems associated with these query languages – the evaluation problem and the containment problem. We isolate the complexity of evaluation at every order, in an analysis similar to that for that standard typed lambda calculus. We show that the containment problem (and hence, the equivalence problem) is decidable in several important subcases, particularly in the case where query constants and variables range over the positive relational operators. The main decidability result relies on techniques that differ from those used in classical query containment. We also show that the analysis of higher-order queries is closely connected to the evaluation and containment problems for non-recursive Datalog.

## 1. Introduction

The basic evaluation and static analysis problems for database queries have been an object of considerable study. For static analysis, the starting point is the seminal work of Chandra and Merlin [13] on containment for conjunctive queries. Algorithms for containment represent a first step in the study of optimization of queries. Outside of relational queries without negation, containment has been little understood. For example, a full understanding even of the decidability line for containment is lacking for languages with complex values, such as bags and lists.

This work focuses on the addition of higher-order features to enhance the expressiveness of traditional relational languages. The motivation for higher-order querying is partially due to the desire of having formalisms that can combine query processing and query transformation – for example, formalisms

*Email addresses:* michael.benedikt@cs.ox.ac.uk (Michael Benedikt), gabriele.puppis@labri.fr (Gabriele Puppis), qhuyvu@gmail.com (Huy Vu)

that can express query rewriting, in which queries over a view are rewritten to queries over base data, or query relaxation [32, 3], where queries are rewritten to get a larger class of results. The connections between higher-order querying and query rewriting are discussed further below, in Section 2. Another application for higher-order querying is query specification [35, 57, 12], which can be seen as Boolean querying of queries. Query specification is an approach to specify permitted queries to secure access to web datasources. The filter can be seen as a higher-order query that takes an input query $Q$ and returns true exactly when $Q$ is permitted for evaluation.

The addition of higher-order functions has been considered in combination with complex values and other structured datatypes, such as XML. For example, a functional approach to querying plays a role in the complex-valued language Monad Algebra introduced by Breazu-Tannen, Buneman, and Wong [49]. The use of functional languages in dealing with complex-valued and XML-based data models has a number of advantages, including better integration with general-purpose functional languages. Since functional languages support higher-order functions, this has led to the addition of higher-order features in the query languages, such as support for higher-order transformation in XQuery 3.0 [43].

Here we consider the interaction of higher-order functions and relational queries in isolation, studying a higher-order query language consisting of terms that have variables ranging over queries as well as variables ranging over relations. Terms are built up via the standard operations of Relational Algebra, plus a new operation, "application", which maps a query variable to an expression. Higher-order terms can be considered in two ways: as functions of the first-order and second order variables together, or (via currying the second-order variables) as mappings from queries to queries.

**Example 1.** *We consider transformations that receive input queries $P, Q$, where both $P$ and $Q$ take as input relations $R$ with integer-valued attributes $a$ and $b$, and return relations of the same type. One such transformation takes $P$ and $Q$ and returns the query $\sigma_{a=5}(P \cap Q)$. This would be expressible in our language as $\lambda \mathtt{P}.\, \lambda \mathtt{Q}.\, \lambda \mathtt{R}.\, \sigma_{a=5}\big(\mathtt{P}(\mathtt{R}) \bowtie \mathtt{Q}(\mathtt{R})\big)$. Another such transformation takes $P$ and $Q$ and returns the query $\sigma_{a=5} \circ Q \circ P$. This would be expressible in our language as $\lambda \mathtt{P}.\, \lambda \mathtt{Q}.\, \lambda \mathtt{R}.\, \sigma_{a=5}\big(\mathtt{Q}(\mathtt{P}(\mathtt{R}))\big)$.*

In the above example, P and Q are *query variables* while R is a relational variable – R ranges over finite databases for a schema with attributes $\{a, b\}$, while P and Q range over mappings between such databases. An important property of these transformation languages is that they are *generic*, in the sense that the output of a term when the higher-order variables are bound to queries depends only on the semantics of the query. This is in contrast to query transformation and specification languages based on syntax, such as those used for query specification in [35, 57, 12]. Thus our languages are in line with the approach to higher-order transformation given in functional programming languages.

Of course, an important question is exactly what mappings P and Q range over, and what Relational Algebra operators are permitted in addition to application. This will depend on the problems that we study.

The most fundamental problem for a query language is evaluation, which is usually defined to be computation of the result of a query on an input database. We will give a comprehensive look at the evaluation problem, looking at a large set of operators, including *all* relational operators and in addition a fixpoint operator. We will restrict to queries that evaluate to a relational instance (i.e. a table), where the output is given explicitly as a set of tuples. This will allow a direct comparison with the query evaluation problem for traditional relational query languages. In contrast, evaluation of queries that return higher type operators is not as clearly defined, since it depends on the representation of the output. The first part of this paper consists of a comprehensive analysis of the evaluation problem, including both upper and lower bounds. We parameterize the complexity depending on the *degree* of the query, which describes the maximal nesting of function types within intermediate outputs.

The second problem we study is that of containment. Bounds on containment immediately give bounds on the equivalence problem. That is, the problem of testing whether two queries always give the same output. Just as evaluation is the fundamental runtime problem, equivalence is arguably the fundamental static analysis problem. Equivalence and containment are known to be undecidable for fragments of ordinary relational query languages in the presence of negation – e.g. for relational algebra including the difference operator. We will thus focus on *negation-free languages* here, restricting the operators appearing in queries to relational algebra without difference or quotient. For these "positive" variants of the higher-order query languages we investigate the complexity of the containment problem. We first isolate the complexity for transformations acting on ordinary relations, via a reduction to results on containment for traditional logic-based query languages. For transformations that are higher-order, our main result is the decidability of containment in the case of transformations over queries ranging over Positive Relational Algebra. We further obtain a precise bound on the complexity of such transformations, provided that the transformations are in a normal form.

Our contributions can be summarized as follows:

- We define a higher-order query language for which many basic analysis problems are decidable, and compare its expressiveness and succinctness with that of existing query languages (Section 4).
- We isolate the complexity of the evaluation problem for the higher-order query language (Section 5).
- We determine the complexity of containment and equivalence for higher-order terms for terms that evaluate to a query (Section 6).
- We isolate the complexity of containment and equivalence for terms that evaluate to a query transformation in several important cases. In particular, we are able to do this in the case where terms are in normal form and manipulate queries of the Positive Relational Algebra (Section 6).

Some of the results here appeared in preliminary form in the conference papers [8] and [59]. This version contains a full semantics of higher-order queries, as well as detailed proofs of the main results.

## 2. Related Work

The languages presented here have not been studied in the prior literature. We overview here the nearest related works, and their distinction from our work.

**Integration of higher-order features in a database query language.** There is a line of research from the 90's in functional databases [27], aiming at the unification of database query languages with functional programming. Kannelakis and collaborators investigated embeddings of relational query languages into typed $\lambda$-calculi [27, 25, 26]. For a good overview of the state of the art at that time, see the Ph.D. thesis of Hillebrand [24]. The goal is to encode the operational semantics of relational query languages in the standard reduction operations of the host calculus. Hillebrand et al. [27] give polynomial time encodings of standard languages, including query languages with recursion mechanisms, within variants of the $\lambda$-calculus. Databases are encoded in terms, using a particular encoding. They deal with both a strongly-typed version of the calculus, and a polymorphic version (see section 2.1 of [24]). In particular, they show that a standard object-oriented calculus can be embedded into the polymorphic version of the calculus.

Compared to the work of Kannelakis and collaborators [27, 25, 26], our languages are designed with a different approach. We do not encode standard query languages using variants of the $\lambda$-calculus. We simply combine queries and $\lambda$-calculus: ordinary queries are treated as fixed constants, with their usual semantics, and we deal with database instances as constants, not via any encodings. Our results are orthogonal to those in prior work in a very strong sense: their complexity results are about terms that code queries (a subset of $\lambda$-terms) and isolate the *data complexity* of such terms. Our results are about *all* terms, and concern the combined complexity, with the lower bounds holding for *query complexity*. Indeed, the data complexity of the query languages we study is always polynomial time. Our results show that the impact of database query constants is localized to low degrees (roughly, to degree equal to the max order of the constants). In fact, we expect that our results could be extended to give a bound on a calculus over arbitrary constants in terms of the complexity of the constants, but we have not explored (or seen) a formalization of this.

On the more practical side, languages such as Machiavelli [40] and Kleisli [60] embed database operations in a general-purpose functional language (ML in both cases above). The type system of the host language is extended with type constructors for various relational and object-oriented database features: e.g., records, variant records, and sets. Higher-order functions can be formed and applied using the constructs of the host language; in particular, the type system can constrain the domain and range of a function on database instances, but the computational power of such functions is limited only by the host language.

Cooper [15] defined a higher-order language that integrates $\lambda$-calculus with nested relational calculus. In his work, he also provides a type-and-effect system for the higher-order language and a translation from the language into SQL. Tackling a similar problem from a practical side, Ulrich [50] has described an implementation that uses the FERRY framework [23] to translate a subset of the LINKS programming language [16], which is functional and strongly typed, into SQL. The FERRY framework explores subsets of programming languages that can be transformed into queries executable by relational database engines.

The lower-order terms of these languages do not match ordinary queries, as in our case, because they do not define sets of values as ordinary queries as we do. More importantly, these authors do not study the complexity of the related computational problems.

**Higher order features and structured datatypes.** Tannen et al. [49] present the nested query language Monad Algebra, using a $\lambda$-calculus over a type system capturing nested relational structures. However, abstraction is allowed only over values (nested data), not over functions. Koch [31] has shown that these languages are equivalent (modulo coding issues) to the functional XML query language XQuery. The expressive power of queries that can arise in a nested relational language is thus bounded: for example, the well-known conservativity theorem of Paredaens and Van Gucht [42] implies that the expressive power of such a language on relational data is no more than that of relational calculus.

As mentioned above, Monad Algebra (as Core XQuery) does not allow abstraction over queries. We consider higher-order functions in a more extensive way, allowing abstraction over queries, queries over queries, etc. We use the simply-typed $\lambda$-calculus with built-in constants, including queries and complex values or XML data, as the framework.

Second, the succinctness of our languages and Monad Algebra are not comparable. Our lower degree terms are much weaker than Nested Relational Algebra (NRA) expressions; they correspond merely to First-Order logic with let bindings, which can be converted tractably to ordinary Relational Algebra expressions (on models of size $> 1$ [4]). Koch [31] has shown (modulo complexity-theoretic assumptions) that this cannot be done for Nested Relational Algebra terms. On the other hand, our higher-order terms over Relational Algebra are not efficiently translatable to NRA terms: they can check for the existence of a doubly-exponential sized path in a graph. In contrast, it follows from [7] that positive Monad Algebra terms can be converted in exponential time to flat existential First-Order queries. Using games one can derive that such terms cannot check for doubly-exponential sized paths.

**Meta-data querying.** Several researchers have looked at the issue of uniformly handling data and meta-data within a query language – particularly, see [34, 39, 14, 44, 45]. The emphasis in most of these works is on queries that include relation names and column information in the input, and output, in manipulating relational queries. An exception is the work of Neven et. al. in [39], which gives a language that can manipulate tables containing both queries

and data.

Our languages are incomparable with these languages, since we do not query the schema and syntax of queries, and they do not allow construction of query transformations. These languages are much more powerful than ours, and extend standard query languages in an intuitive way. But they do not satisfy either of our two design goals, since they are relationally complete and allow one to access the syntactic structure of queries.

**Query specification.** One of our motivations is query specification, where one wishes to restrict the queries that can be run by sources [35, 57, 12].

Vassalos and Papakonstantinou introduce description languages to describe all possible queries that the sources can produce [57]. A description language, called p-Datalog (Parameterized Datalog), is similar to Datalog but has a set of symbols called tokens. Tokens are variables but when a query is created, all tokens have to be instantiated. Levy et al. introduced a method which uses Datalog programs to encode families of views [35]. First, the set of views is partitioned into a finite number of equivalence classes from which representatives can be chosen. Given a query $Q$, by defining equivalent views, the set of views is encoded by a Datalog program. Cautis et al. [12] have studied the expressibility and support for querying an infinite number of sources generated by a Datalog program under dependencies. They show that the expressibility problem and the support problem are inter-reducible in polynomial time. When source constraints are included, [12] gives restricted cases such that the problem can be reduced to the dependency-free problem. When considering the decidability under a mix of key and weakly acyclic foreign key constraints, the Datalog program can be restricted to have decidability results. The restrictions allow us to pre-process the constraints and reduce to the dependency-free case which is shown in [35].

As with meta-data querying, the distinction from our work is that we query transformation only via their semantics, not via their syntax, in line with the mechanisms of standard functional languages.

**Related work on the evaluation and containment problems.** Evaluation of relational queries is a heavily-studied problem. A myriad of results, including the basic complexity bounds, can be found in [1]. Full Relational Algebra, for instance, is known to have a PSPACE-complete evaluation problem. Koch [31] has shown a NEXP lower bound and an EXPSPACE upper bound on the evaluation problem for a fragment of XQuery and also for Monad Algebra. We use techniques from [31] in our lower bound results here.

Evaluation of $\lambda$-calculus expressions also has a long history. Statman [48] proved that typed $\lambda$-calculus is non-elementary. A simple proof of Statman's result is later given by Mairson [37]. The hyperexponential time and space complexity of evaluating higher-order queries as a unification of database query languages with functional programming has been shown by Hillebrand and Kanellakis [26].

Query equivalence and containment have been studied extensively for many relational query classes: e.g., conjunctive queries and union of conjunctive

queries, starting with [13]. Benedikt and Gottlob [6] have isolated the complexity of containment between two nonrecursive Datalog programs. There is also work for Nested Relational Algebra and other complex object models. Levy and Suciu [36] investigate containment and equivalence between queries on complex objects. Later work of Dong et al. [18] studies the containment problem for nested XML queries, while Björklund et al. [9] have shown a full picture of the complexity of the containment problem for conjunctive queries over trees.

## 3. Logic-based query languages

In this section, we briefly recall some standard formalisms based on logic for defining queries on database relations.

**Conjunctive queries.** Conjunctive queries are essentially queries definable in first-order logic using conjunctions, existential quantifications, and relational predicates, including equality, where the relational atoms can include both variables and constants. Here we adopt a rule-based presentation of conjunctive queries. Formally, we define a *conjunctive query* as a rule of the form

$$Q(\bar{\mathtt{x}}) \ :- \ R_1(\bar{\mathtt{y}}_1), \ \ldots, \ R_n(\bar{\mathtt{y}}_n)$$

where $Q$ is the output relation defined by the query, $R_1, \ldots, R_n$ are the input relations, and $\bar{\mathtt{x}}, \bar{\mathtt{y}}_1, \ldots, \bar{\mathtt{y}}_n$ are tuples of variables and constants, with every variable of $\bar{\mathtt{x}}$ occurring at least once in the tuples $\bar{\mathtt{y}}_1, \ldots, \bar{\mathtt{y}}_n$.

Given a database instance consisting of relations $R_1, \ldots, R_n$ of appropriate arities, the result of the above query $Q$ is defined as

$$Q(R_1, \ldots, R_n) \ = \ \left\{ \nu(\bar{\mathtt{x}}) \ \middle| \ \begin{array}{l} \nu \text{ is a valuation for the variables in } \bar{\mathtt{y}}_1, \ldots, \bar{\mathtt{y}}_n \\ \text{such that } \nu(\bar{\mathtt{y}}_i) \in R_i \text{ for all } i = 1, \ldots, n \end{array} \right\} .$$

Intuitively, the result $Q(R_1, \ldots, R_n)$ contains all tuples of values that can be matched to $\bar{\mathtt{x}}$ and that can be extended to a valuation for the variables in $\bar{\mathtt{y}}_1, \ldots, \bar{\mathtt{y}}_n$, so as to satisfy the predicates $R_1(\bar{\mathtt{y}}_1), \ \ldots, \ R_n(\bar{\mathtt{y}}_n)$. We remark that, although there are no quantifiers in the rule-based notation of a conjunctive query, variables appearing in the left-hand side are implicitly quantified universally, and variables only appearing in the right-hand side are implicitly quantified existentially.

**Example 2.** *The conjunctive query* $Q(\mathtt{x}_1, \mathtt{x}_2) \ :- \ R_1(\mathtt{x}_1, \mathtt{y}) \wedge R_2(\mathtt{y}, \mathtt{x}_2)$ *receives as input two relations* $R_1$ *and* $R_2$ *of arity* 2 *and returns their composition, that is, the relation* $Q(R_1, R_2) = \big\{(c_1, c_2) \ \big| \ \exists d. \ (c_1, d) \in R_1, \ (d, c_2) \in R_2\big\}$.

A *union of conjunctive queries* is syntactically given as $Q = \bigcup_{i=1,\ldots,m} Q_i$, where $Q_1, \ldots, Q_m$ are conjunctive queries having the same arity on the output predicates. The result of applying a union of conjunctive queries $Q = \bigcup_{i=1,\ldots,m} Q_i$ to a tuple $\bar{R}$ of database relations is naturally defined as

$$Q(\bar{R}) \ = \ \bigcup_{i=1,\ldots,m} Q_i(\bar{R}) \ .$$

**Comparison with algebra-based languages.** The syntax of the higher order languages we will define are based on extending relational algebra, not logic-based syntax. However we will compare our languages with logic-based ones like those above, so we recall some classical results comparing algebra-based and logic-based approaches (again, a good reference is [1]).

We recall that expressions of Relational Algebra are built up from input and constant relations using the operations of selection $\sigma_\varphi$, projection $\pi_A$, join $\bowtie$, attribute renaming $\rho_{a/b}$, union $\cup$, and set difference $\setminus$. The queries definable by expressions of Relational Algebra are clearly more expressive than conjunctive queries. However, the fragment of Relational Algebra that only uses constant singleton relations and the operations of selection, projection, join, and renaming describes precisely the conjunctive queries. For example, the query $Q(\mathtt{x}_1, \mathtt{x}_2) \ :- \ R_1(\mathtt{x}_1, \mathtt{y}) \wedge R_2(\mathtt{y}, \mathtt{x}_2)$ of Example 2 can be equally described by the expression of Relational Algebra $\rho_{b/c}(R_1) \bowtie \rho_{a/c}(R_2)$, assuming that the relations $R_1$ and $R_2$ are over the attributes $(a, b)$. Similarly, unions of conjunctive queries can be equally described by Relational Algebra expressions that avoid the operation of set difference.

**Datalog.** Another formalism that will be used in the paper is that of Datalog programs. Specifically, a *Datalog program* consists of a set of predicates partitioned into *extensional predicates* and *intensional predicates*, a distinguished intensional predicate called the *goal predicate*, and a set of rules that define the intensional predicates. Rules are of the form

$$R(\bar{\mathtt{x}}) \ :- \ S_1(\bar{\mathtt{y}}_1), \ \ \ldots, \ S_n(\bar{\mathtt{y}}_n)$$

where $R$ is an intensional predicate, $S_1, \ldots, S_n$ are predicates, and $\bar{\mathtt{x}}, \bar{\mathtt{y}}_1, \ldots, \bar{\mathtt{y}}_n$ are tuples of variables and constants, with every variable of $\bar{\mathtt{x}}$ occurring at least once in the tuples $\bar{\mathtt{y}}_1, \ldots, \bar{\mathtt{y}}_n$. Intuitively, each rule of the above form can be seen as a conjunctive query that adds tuples to the predicate $R$ on the basis of the tuples from the predicates $\bar{\mathtt{y}}_1, \ldots, \bar{\mathtt{y}}_n$ and the equality patterns between variables and constants in $\bar{\mathtt{x}}, \bar{\mathtt{y}}_1, \ldots, \bar{\mathtt{y}}_n$. Note that, as special cases, we can have rules of the form $R(\bar{c}) \ :-$ , with no variables and empty right-hand side, which state that predicate $R$ contains tuple $\bar{c}$. Also note that multiple rules with the same left-hand side can be used in a Datalog program.

Datalog programs define queries whose input is an instance for the extensional predicates, and whose output is an instance of the goal predicate. All of the intensional predicates are computed via a fixpoint process starting with the empty relation. At every step of the fixpoint process, the rules are applied to form the next iteration of each intensional predicate.

In the special case where there are no extensional predicates, a Datalog program simply produces an instance of the goal predicate. We refer to this as an *input-free* Datalog program.

**Example 3.** *The following is an example of a Datalog program that computes the transitive closure of an input relation $R$ of arity 2 (the intensional goal*

*predicate is $Q$):*

$$Q(\mathtt{x}_1, \mathtt{x}_2) \ :- \ R(\mathtt{x}_1, \mathtt{x}_2)$$
$$Q(\mathtt{x}_1, \mathtt{x}_2) \ :- \ Q(\mathtt{x}_1, \mathtt{y}), \ Q(\mathtt{y}, \mathtt{x}_2) \ .$$

We conclude by recalling the definitions of some interesting variants of Datalog. *Non-recursive Datalog* restricts the dependency relation among intensional predicates to be acyclic. That is, the intensional predicates are layered, with the predicates of layer $i$ defined by rules referring only to extensional predicates and intensional predicates in layers below $i$. *Datalog with Stratified Negation* allows more general rules of the form

$$R(\bar{\mathtt{x}}) \ :- \ \phi_1(\bar{\mathtt{y}}_1), \ \ldots, \ \phi_n(\bar{\mathtt{y}}_n)$$

where the $\phi_i$ are either relational atoms or their negations. The intensional predicates are again grouped into strata, and rules involving a predicate of layer $i$ can use a negated atom only with an intensional predicate of a lower layer.

We refer to [1] for additional details concerning the syntax and semantics of the above variants of Datalog.

## 4. Higher-order queries

This section defines the higher-order query language, called HO, which is the subject of the remaining sections. We will also define a particularly simple, expressively equivalent subset of the language – the normal form queries.

The higher-order language HO is an extension of Relational Algebra based on the concepts of abstraction and application. It contains constants for the database relations and the operators of Relational Algebra. Its syntax is defined according to the following grammar:

$$\mathsf{HO} \ := \ \mathtt{const} \mid \mathtt{X} \mid \lambda \mathtt{X}.\, \mathsf{HO} \mid \mathsf{HO}(\mathsf{HO})$$

where `const` denotes a constant name from a fixed (possibly infinite) set, called *signature*, and `X` denotes a variable name from a fixed infinite set. For example, the constants in the signature may include some relation names and the usual operators of Relational Algebra:

$$\mathtt{const} \ := \ R \mid \sigma_\varphi \mid \rho_{a/b} \mid \pi_A \mid \bowtie \mid \cup \mid \setminus$$

where $R$ is a relation name, $\varphi$ is a condition, $a$ and $b$ are attribute names, and $A$ is a set of attribute names. Below, we formally define the types and the semantics for this language.

**Relational types.** We fix an infinite set of *attribute names* (or *attributes*). We associate with each attribute $a$ a range $\mathsf{rng}(a)$ of possible values, called the *attribute range* of $a$. We will usually let the range of our attributes to be the set $\mathbb{Z}$ of integers.

The base types are the *relational types*, each given by a (possibly empty) tuple $\tau = (a_1, \ldots, a_m)$ of pairwise distinct attribute names. The *arity* of a relational type $\tau = (a_1, \ldots, a_m)$ is the number $m$ of attribute names in it. We manipulate relational types by using standard operations such as the juxtaposition $(\tau, \tau')$ of two types and the restriction $\tau|_A$ of a type to a subset $A$ of its attributes.

As usual, a *tuple* for a relational type $\tau = (a_1, \ldots, a_m)$ is a function from the attribute names $a_i$ to the attribute ranges $\mathsf{rng}(a_i)$. An *instance* of a relational type is a set of tuples. The *domain* of a relational type $\tau$ is the collection of all finite instances of $\tau$. Sometimes, particularly in queries, instead of attribute names we will use the positional notation (according to the fixed ordering on attributes) for addressing the elements of a tuple. For instance, given a tuple $\bar{c}$ of type $(a_1, \ldots, a_m)$, we can write $\bar{c}[i]$ to denote $\bar{c}(a_i)$.

Although we do not consider boolean attributes here, we do have a "boolean type" – the type with no attributes, denoted $()$. Note that there are only two instances of type $()$, namely, the empty instance, which we identify with the boolean value $\mathsf{false}$, and the set $\{\varepsilon\}$ that consists of a single empty tuple, which we identify with the boolean value $\mathsf{true}$.

**Higher-order types.** Relational types are the basic building blocks of more complex types. We define *higher-order types* by using the functional type constructor: if $\tau_1, \tau_2$ are types with domains $D_1 = \mathsf{dom}(\tau_1)$ and $D_2 = \mathsf{dom}(\tau_2)$, then $\tau_1 \to \tau_2$ is a type with domain $D_2^{D_1}$, which contains all functions from $D_1$ to $D_2$,

Every type can be uniquely written as $\tau_1 \to \tau_2 \to \ldots \to \tau_k \to \tau$, where $\tau$ is a relational type and arrows associate to the right; this type is sometimes abbreviated to $(\tau_1 \times \ldots \times \tau_k) \to \tau$ (note that the latter is only an abbreviation, since we have no product operation on types). The *order* $\mathsf{order}(\tau)$ of a type $\tau$ is a natural number that is inductively defined as follows. All relational types have order 0, all functional types $\tau_1 \to \tau_2$ have order $\mathsf{max}\big(\mathsf{order}(\tau_1) + 1,\ \mathsf{order}(\tau_2)\big)$. Order-1 types are often called *query types*.

**Simply typed terms.** We will assign types to terms of $\mathsf{HO}$ according to the following typing rules:

- every constant or variable of type $\tau$ is a term of the same type $\tau$;
- if $\mathtt{X}$ is a variable of type $\tau_1$ and $t$ is a term of type $\tau_2$, then $\lambda\mathtt{X}.\ t$ is a term of type $\tau_1 \to \tau_2$;
- if $t$ is a term of type $\tau_1 \to \tau_2$ and $u$ is a term of type $\tau_1$, then $t(u)$ is a term of type $\tau_2$.

We will always assume that our terms are *well-typed*, in that a type can be assigned by the rules above. We say that a term $t$ is *closed* if it does not contain free variables, namely, if all variable occurrences in $t$ are under the scope of an abstraction operator.

The *order of a term* $t$, denoted $\mathsf{order}(t)$, is the order of its type. Besides the order, we associate with each term $t$ another number: the *degree of* $t$, denoted

$\mathsf{degree}(t)$ and defined as the *maximum order of the variables* that appear in $t$. For instance, if $\mathtt{R}$ is a relational variable of type $\tau = (a)$ and $\sigma_{a=0}$ is a query constant of type $\tau \to \tau$, then the term $\lambda\mathtt{R}.\ \sigma_{a=0}(\mathtt{R})$ is a term of type $\tau \to \tau$, order 1, and degree 0. Intuitively, the order of a term measures the nesting of abstractions in the denoted object, while the degree measures the complexity of the representation itself (the same relation, query, etc. can be represented by means of terms of different degrees).

We also define the *size* of a term inductively as follows. The size of a variable is the size of a standard string representation of its type. The size of a relational constant is the size of the corresponding instance, namely, the number of its attributes times the number of rows. The size of a query constant is the size of its standard string representation – for instance, the size of a projection operator $\pi_A$ is 1 plus the length of the string needed to represent the attribute names in $A$; the size of the remaining query constants of Relational Algebra should be clear. Finally, the size of a higher-order term is inductively defined as 1 plus the sum of the sizes of its top-level subterms.

We introduce the following notation, which will be used through the rest of the document.

**Definition 4.** *Let $\Sigma$ be a generic signature and let $o, d \in \mathbb{N}$. We denote by*

$$\mathsf{HO}_o^d[\Sigma]$$

*the class of all closed terms of order $o$ and degree at most $d$ that are built up from constants in the signature $\Sigma$. We further let $\mathsf{HO}[\Sigma] = \bigcup_{o,d\in\mathbb{N}} \mathsf{HO}_o^d[\Sigma]$.*

For example, if $\Sigma$ is the signature containing the operators of Relational Algebra, then $\mathsf{HO}_1^0[\Sigma]$ contains the query term $t = \lambda\mathtt{R}.\ \pi_{a,b}\big(\rho_{b/c}(\mathtt{R}) \bowtie \rho_{a/c}(\mathtt{R})\big)$, but not the query term $t' = \big(\lambda\mathtt{Q}.\ \lambda\mathtt{R}.\ \mathtt{Q}(\mathtt{Q}(\mathtt{R}))\big)(\sigma_\varphi)$, which has degree 1 and hence belongs to $\mathsf{HO}_1^1[\Sigma]$.

**Semantics.** Here we give a denotational semantics for our $\mathsf{HO}$ terms, which is essentially that of the simply typed $\lambda$-calculus extended with the interpretation of relational and query constants.

In order to define this semantics, we need to first specify an interpretation for the constants. Formally, an *interpretation* for a signature $\Sigma$ is a function $\iota$ that maps constants in $\Sigma$ to concrete instances over the corresponding domains. As an example, an interpretation $\iota$ could map the query constant $\cup$ to the function that receives two database relations of the same type and returns their union.

Below, we provide a formal definition of the semantics, which is naturally given by an induction on the order of the terms. In doing so, we make the interpretation $\iota$ explicit by denoting with $[\![t]\!]_\iota$ the semantics of a term $t$.

**Definition 5.** *Let $\iota$ be an interpretation for a signature $\Sigma$ and let $t$ be a closed term in $\mathsf{HO}[\Sigma]$. The semantics $[\![t]\!]_\iota$ is defined by induction on the order of $t$:*

- *If $t$ is a constant from $\Sigma$, then $[\![t]\!]_\iota = \iota(t)$.*

- *If $t$ is an abstraction of the form $\lambda \mathtt{X}.\, t_2$ of type $\tau_1 \to \tau_2$, then $[\![t]\!]_\iota$ is the function from $\mathsf{dom}(\tau_1)$ to $\mathsf{dom}(\tau_2)$ that maps any object $O$ to the object $[\![t_2]\!]_{\iota[\mathtt{X} \mapsto O]}$, where $t_2$ is seen as a closed term over the signature $\Sigma$ extended with the constant $\mathtt{X}$, and $\iota[\mathtt{X} \mapsto O]$ is the interpretation for the new signature that extends $\iota$ by mapping $\mathtt{X}$ to $O$.*

- *If $t$ is an application of the form $t_2(t_1)$, then $[\![t]\!]_\iota = [\![t_2]\!]_\iota\big([\![t_1]\!]_\iota\big)$.*

As an example, if $\mathtt{Q}$ is a query variable of type $(a) \to (a)$, with $\mathsf{rng}(a) = \mathbb{Z}$, then the semantics $[\![t]\!]$ of the term $t = \lambda \mathtt{Q}.\, \mathtt{Q}\big(\{1\}\big)$ is the function that maps any query $Q : 2^{\mathbb{Z}} \to 2^{\mathbb{Z}}$ to the set $Q\big(\{1\}\big)$.

**Specific signatures.** We will mainly consider signatures with constants of type order 0 or 1, that is, with *relational constants* and *query constants*. In particular, unless otherwise specified, all signatures have no constants of order greater than or equal to 2 (the signature $\mathsf{IFP}$ that is defined below will be the only exception). We will also assume that the signatures have constants for all database relations (the only exception being the signature $\mathsf{SPJ}^{\mathsf{sing}}$, defined below, that contains relational constants only for the *singleton* database relations).

Recall that we defined the semantics of higher-order terms with respect to a given signature $\Sigma$ and a given interpretation $\iota$ for the constants in it. From now on, we tacitly assume the standard interpretation of the query constants $\sigma_\varphi$, $\pi_A$, $\rho_{a/b}$, $\bowtie$, $\cup$ of Relational Algebra that may appear in the signature $\Sigma$. Despite the difference between a constant symbol and its interpretation, we will often abuse notation by denoting them in the same way, that is, we will often write $q$ in place of $[\![q]\!]$ for the standard interpretation of a query constant $q$. Similarly, the interpretation of a relational constant is determined by its name, like, for instance, in $\big\{(1,2),(3,4)\big\}$. This allows us to omit the subscript $\iota$ for the underlying interpretation when denoting the semantics of a term.

We will mainly focus on the following specific signatures:

- $\mathsf{RA}^+$ denotes the signature that contains all relational constants plus the operators of Positive Relational Algebra, namely, the selection operator $\sigma_\varphi$, where $\varphi$ is any conjunction of equalities over attributes and constants, the projection operator $\pi_A$, where $A$ is any set of attributes, the renaming operator $\rho_{a/b}$, where $a, b$ are attributes, the join operator $\bowtie$, and the union operator $\cup$. All operators in $\mathsf{RA}^+$ receive one or two relation(s) as input, and return a single relation as output. Thus, they all have types of order 1. In fact, there exist copies of these operators for all possible types of input relations. Since the semantics of the operators is clear, we only give the types for them (in the following, $\tau, \tau_1, \tau_2$ denote generic relational types). A selection operator $\sigma_\varphi$ has type of the form $\tau \to \tau$. A projection operator $\pi_A$ has type of the form $\tau \to \tau|_A$, with $A$ subset of $\tau$. A renaming operator $\rho_{a/b}$ has type of the form $\tau \to \tau_{a/b}$, with $\tau_{a/b}$ obtained from $\tau$ by replacing the attribute name $a$ with a new attribute name $b$. A join operator $\bowtie$ has type of the form $(\tau_1 \times \tau_2) \to \tau$ (or, equally, $\tau_1 \to \tau_2 \to \tau$), with $\tau = (\tau_1, \tau_2)$. A union operator $\cup$ has type of the form $(\tau \times \tau) \to \tau$.

- $\mathsf{RA}$ extends the signature $\mathsf{RA}^+$ with difference operators $\setminus$, again parameterized by a relational type $\tau$ and having order-1 type $(\tau \times \tau) \to \tau$.
- $\mathsf{IFP}$ further extends the signature $\mathsf{RA}$ by adding order-2 constants $\mathsf{ifp}$ of type $(\tau \to \tau) \to \tau \to \tau$, for each relational type $\tau$. These constants are interpreted by second-order functions that map any query $Q$ of Relational Algebra of type $\tau \to \tau$ and any relation $R$ of type $\tau$ to the *inflationary fixpoint of $Q$ on $R$*, namely, to the relation $\bigcup_{n\in\mathbb{N}} R_n$, where $R_0 = R$ and $R_{n+1} = R_n \cup Q(R_n)$ – note that the fixpoint is provably a finite relation.
- $\mathsf{SPJ}$ is the signature obtained from $\mathsf{RA}^+$ by removing unions, that is, $\mathsf{SPJ}$ contains all relational constants and the operators $\sigma_\varphi$, $\pi_A$, $\rho_{a/b}$, and $\bowtie$.
- $\mathsf{SPJ}^{\mathsf{sing}}$ further restricts the signature $\mathsf{SPJ}$ by removing all non-singleton relational constants. For example, $\mathsf{HO}[\mathsf{SPJ}^{\mathsf{sing}}]$ contains the term $t = \lambda \mathtt{R}.\ \mathtt{R} \bowtie \{(1,2)\}$, but not the term $t' = \lambda \mathtt{R}.\ \mathtt{R} \bowtie \{(1,2),(3,4)\}$. In general, order-1 terms over this signature $\mathsf{SPJ}^{\mathsf{sing}}$ capture precisely the conjunctive queries.

**Normal form.** We now recall the notions of $\alpha$-conversion, $\beta$-reduction, and ($\beta$-)normal form from [28]. We will write $t \to t'$, and will say that $t$ *reduces to* $t'$, whenever term $t$ can be rewritten into term $t'$ by using the reduction rules described below. The first reduction rule is that of *$\alpha$-conversion*, which replaces a bound variable $\mathtt{X}$ with a fresh variable $\mathtt{Y}$:

$$\frac{\mathtt{Y} \text{ does not occur in t}}{\lambda \mathtt{X}.\ t \ \to\ \lambda \mathtt{Y}.\ t[\mathtt{X}/\mathtt{Y}]} \qquad (\alpha\text{-conversion})$$

The second rule is that of *$\beta$-reduction*, which substitutes (in a capture-avoiding manner) a formal parameter of a term with the corresponding argument:

$$\frac{\text{no free variable of } t_1 \text{ is bound in } t_2}{(\lambda \mathtt{X}.\, t_2)\,(t_1) \ \to\ t_2[\mathtt{X}/t_1]} \qquad (\beta\text{-reduction})$$

In the above rule, the subterm $(\lambda \mathtt{X}.\ t_2)\, t_1$ is called *redex*. We naturally extend reduction rules to subterms, as follows:

$$\frac{t \ \to\ t'}{\lambda \mathtt{X}.\, t \ \to\ \lambda \mathtt{X}.\, t'} \qquad \frac{t_1 \ \to\ t_1' \qquad t_2 \ \to\ t_2'}{t_2(t_1) \ \to\ t_2'(t_1')} \qquad (\text{context})$$

The reduction rules that we just described are *sound* with respect to the denotational semantics of $\mathsf{HO}$ terms, namely, if $t, t'$ are two terms such that $t \to t'$, then $[\![t]\!] = [\![t']\!]$. In particular, reduction rules preserve the order of a term, but do not preserve in general its degree.

We say that a term $t$ is in *normal form* if it contains no redex (and hence no $\beta$-reduction can be applied to it). For example, the term $\pi_A(R_0)$ is in normal form, while $\big(\lambda \mathtt{R}.\ \pi_A(\mathtt{R})\big)(R_0)$ is not. Since the operation of $\beta$-reduction is confluent and always terminating (on well-typed terms) [21], we have that

every term $t$ can be transformed by a series of $\beta$-reductions to a term that is in normal form and is *unique up to $\alpha$-conversion*. We denote by $t^{\downarrow}$ the normal form of a term $t$. We further denote by

$$\mathsf{HO}_o^{\downarrow}[\Sigma]$$

the class of all closed terms over the signature $\Sigma$ that have order $o$ and are in normal form.

We observe that if $t$ is a closed term of relational type, then its normal form $t^{\downarrow}$ is an expression in $\mathsf{HO}_0^{\downarrow}[\Sigma]$ that contains only relational and query constants, and no variables. This expression can be evaluated, using the semantics of the constants to get a relation. Thus we have a (naive but) effective way of evaluating closed terms of relational type. In Section 5 we will see that the operational semantics obtained by combining $\beta$-reduction with the standard interpretation of relational and query constants is sufficient to obtain evaluation algorithms for terms of arbitrary high degree.

We can also see how the ordinary relational calculus embeds in our language. Closed terms of $\mathsf{HO}_1^{\downarrow}[\mathsf{RA}]$ correspond exactly to the query expressions that can be formed in Relational Algebra. Similarly, closed terms of $\mathsf{HO}_1^{\downarrow}[\mathsf{RA}^+]$ correspond to the query expressions of Positive Relational Algebra, and closed terms of $\mathsf{HO}_1^{\downarrow}[\mathsf{SPJ}]$ correspond to select-project-join queries with relational constants [1]. Moreover, closed terms in normal form which are built up using *singleton relational constants* and query constants of $\mathsf{RA}^+$, and in which unions appear only at the *topmost levels*, translate efficiently to unions of conjunctive queries.

**Relationship with Datalog.** As we mentioned earlier, query terms in normal form correspond to simple query expressions of (fragments of) Relational Algebra. Below we show that the unnormalized terms of degree 0 are also familiar objects in database querying, namely, they correspond to Datalog programs.

**Proposition 6.** *There exist polynomial-time translations between:*

- $\mathsf{HO}_0^0[\mathsf{RA}^+]$ *and Input-free Non-recursive Datalog,*
- $\mathsf{HO}_0^0[\mathsf{RA}]$ *and Input-free Non-recursive Datalog with Stratified Negation,*
- $\mathsf{HO}_0^0[\mathsf{IFP}]$ *and Input-free Datalog with Stratified Negation,*
- $\mathsf{HO}_0^0[\mathsf{SPJ}^{\mathsf{sing}}]$ *and Input-free Non-recursive Datalog in which every intentional predicate occurs on the left-hand side of at most one rule.*

*Similar translations exist between terms of order* 1 *and Datalog programs with input.*

*Proof.* We only provide the translations for terms of order 0 over the signature $\mathsf{RA}^+$; the translations for the remaining items can be easily derived.

For the first direction, we show how to define all predicates of a Non-recursive Datalog program by means of suitable relational terms in $\mathsf{HO}_0^0[\mathsf{RA}^+]$. For this we use induction based on the (acyclic) dependency relation between predicates.

In the base case, we consider a Datalog program and an extensional predicate $R$ in it that is defined by a set of facts $R(\bar{c}_1)$, ..., $R(\bar{c}_\ell)$, and we accordingly construct the equivalent term

$$t_R \;=\; \{\bar{c}_1, \ldots, \bar{c}_\ell\}.$$

In the inductive step, we use as a basic building block the well-known correspondence between Datalog clauses and simple expressions of Positive Relational Algebra – which are thus trivially normalized query terms – that is, elements of $\mathsf{HO}_1^{\downarrow}[\mathsf{RA}^+]$. This procedure identifies predicates with relations and positions of formal arguments with attribute names. This classical construction (e.g., [1]) turns any Datalog rule $R(\bar{\mathtt{x}}) \;:\!-\; S_1(\bar{\mathtt{y}}_1),\; \ldots,\; S_n(\bar{\mathtt{y}}_n)$ into an equivalent term $q \in \mathsf{HO}_1^{\downarrow}[\mathsf{RA}^+]$ whose semantics maps a tuple of input relations $S_1, \ldots, S_n$ to the least relation $R$ that satisfies the clause. For example, the Datalog rule on the left is translated to the term on the right:

$$R(\mathtt{x},\mathtt{y}) \;:\!-\; S(\mathtt{x},\mathtt{z}),\; T(\mathtt{z},\mathtt{y}) \qquad\qquad \lambda\mathtt{S}.\;\lambda\mathtt{T}.\;\rho_{\substack{\mathtt{x}/1\\ \mathtt{y}/2}}\left(\pi_{\mathtt{x},\mathtt{y}}\left(\rho_{\substack{1/\mathtt{x}\\ 2/\mathtt{z}}}(\mathtt{S}) \bowtie \rho_{\substack{1/\mathtt{z}\\ 2/\mathtt{y}}}(\mathtt{T})\right)\right)$$

We now return to the inductive process, considering of a predicate $R$ that is defined intentionally by a set of clauses:

$$\begin{aligned} R(\bar{\mathtt{x}}_1) \;&:\!-\; S_{1,1}(\bar{\mathtt{y}}_{1,1}),\; \ldots,\; S_{1,n_1}(\bar{\mathtt{y}}_{1,n_1}) \\ &\vdots \\ R(\bar{\mathtt{x}}_m) \;&:\!-\; S_{m,1}(\bar{\mathtt{y}}_{m,1}),\; \ldots,\; S_{m,n_m}(\bar{\mathtt{y}}_{m,n_m}) \end{aligned}$$

By the induction hypothesis, we can assume that all predicates $S_{1,1}, \ldots, S_{m,n_m}$ appearing in the bodies of the clauses have been translated to equivalent relational terms $t_{S_{1,1}}, \ldots, t_{S_{m,n_m}} \in \mathsf{HO}_0^0[\mathsf{RA}^+]$. Using the correspondence between non-recursive Datalog clauses and simple expressions of Relational Algebra alluded to above, we can construct $m$ terms $q_1, \ldots, q_m \in \mathsf{HO}_1^{\downarrow}[\mathsf{RA}^+]$ equivalent to the $m$ clauses above. We finally construct the term $t_R \in \mathsf{HO}_0^0[\mathsf{RA}^+]$ that equally defines the predicate $R$ as follows:

$$t_R \;=\; \left(\lambda\mathtt{S}_{1,1} \ldots \lambda\mathtt{S}_{m,n_m}.\; \bigcup_{1\le i\le m} q_i(\mathtt{S}_{i,1}, \ldots, \mathtt{S}_{i,n_i})\right)\; (t_{S_{1,1}}) \ldots (t_{S_{m,n_m}}).$$

For the converse direction, we have to transform a generic term $t \in \mathsf{HO}_0^0[\mathsf{RA}^+]$ into an equivalent non-recursive Datalog program. For this we proceed by structural induction on the subterm occurrences of $t$. We let $t_1, \ldots, t_n$ be all the subterms that occur as arguments of redexes of $t$. Without loss of generality, we can assume that whenever $t_i$ contains $t_j$ as a subterm, then $i \ge j$. Furthermore, we define for convenience $t_{n+1} = t$. Note that, since $t$ has degree at most 1, every (sub)term $t_i$ evaluates to a relation $R_i$. By induction on $i \;=\; 1, \ldots, n+1$, we will construct a suitable Datalog program $P_i$ that defines the relation $R_i = [\![t_i]\!]$. Fix $1 \le i \le n+1$ and assume by induction that there exist Datalog programs $P_1, \ldots, P_{i-1}$ defining the relations $R_1 = [\![t_1]\!]$, ..., $R_{i-1} = [\![t_{i-1}]\!]$. We distinguish

two cases. If $t_i$ is a relational constant of the form $\{\bar{c}_1, \ldots, \bar{c}_\ell\}$, then we simply let $P_i$ consist of the facts $R_i(\bar{c}_1)$ :– , …, $R_i(\bar{c}_\ell)$ :– . Otherwise, $t_i$ must be a redex of the form $(\lambda \mathtt{R}.\ q)\ (t_j)$, with $j < i$ and $(\lambda \mathtt{R}.\ q) \in \mathsf{HO}_1^{\downarrow}[\mathsf{RA}^+]$. Using the correspondence between $\mathsf{HO}_1^{\downarrow}[\mathsf{RA}^+]$ and non-recursive Datalog, we can construct a set $Q$ of non-recursive Datalog clauses that is equivalent to the term $q$ on all interpretations of the formal parameter $\mathtt{R}$. In particular, this means that if we replace in $Q$ all occurrences of $\mathtt{R}$ with $R_j$ and we add the rules of $P_j$ (recall that $P_j$ defines the relation $R_j = [\![t_j]\!]$), then we obtain a non-recursive Datalog program that defines precisely the relation $[\![q]\!]_{[\mathtt{R} \mapsto R_j]} = [\![t_i]\!] = R_i$. □

The following example demonstrates the translation from a degree-0 term to a Datalog program.

**Example 7.** *Let* $\mathtt{R}$ *be a variable of relational type* $\tau = (a, b)$ *and consider the following term of* $\mathsf{HO}_0^0[\mathsf{RA}^+]$*:*

$$t \;=\; \big(\lambda \mathtt{R}.\ \mathtt{R} \;\cup\; \pi_{a,b}\big(\rho_{b/c}(\mathtt{R}) \bowtie \rho_{a/c}(\mathtt{R})\big)\big)\ \big(\{(1,2),(2,3)\}\big).$$

*As* $t$ *contains only one redex with a relational constant as argument, it can be translated to the following set of facts and clauses of Datalog:*

$$\begin{array}{ll} R_1(1,2)\ \text{:–} & R_2(\mathtt{x},\mathtt{y})\ \text{:–}\ R_1(\mathtt{x},\mathtt{y}) \\ R_1(2,3)\ \text{:–} & R_2(\mathtt{x},\mathtt{y})\ \text{:–}\ R_1(\mathtt{x},\mathtt{z}),\ R_1(\mathtt{z},\mathtt{y}). \end{array}$$

**Succinctness.** We conclude the section by looking at the succinctness of terms of higher degree. We start by explaining that sharing of subterms can make unnormalized terms much more succinct than their normalized counterparts. In the following example we exploit this to construct 'small' terms in the signature $\mathsf{SPJ}^{\mathsf{sing}}$ that check the existence of 'long' paths inside the graph representation of an input relation (the length of the paths is exponential in the size of the terms).

**Example 8.** *Let* $\mathtt{R}$ *be a variable of relational type* $\tau = (a, b)$ *and let* $\mathtt{Q}$ *be a variable of query type* $\tau \to \tau$*. Consider the following two terms:*

$$\begin{aligned} t_1 &= \lambda \mathtt{R}.\ \pi_{a,b}\big(\rho_{b/c}(\mathtt{R}) \bowtie \rho_{a/c}(\mathtt{R})\big) \\ t_2 &= \lambda \mathtt{Q}.\ \lambda \mathtt{R}.\ \mathtt{Q}\big(\mathtt{Q}(\mathtt{R})\big). \end{aligned}$$

*The term* $t_1$ *receives as input a relation* $R$ *and returns as ouput the composition* $R \circ R = \big\{(x,y) : \exists z.\ (x,z) \in R \wedge (z,y) \in R\big\}$*. The term* $t_2$ *computes the* 2*-fold iteration* $Q^2$ *of an input query* $Q$*. By applying* $t_2$ *to* $t_1$*, one obtains an order-*1 *degree-*1 *term* $t_2(t_1)$ *that receives an input relation* $R$ *and returns all pairs of elements connected by a path in* $R$ *of length exactly* $2^2 = 4$*. Similarly, the term* $t_2(t_2(t_1))$ *is equivalent to two nested applications of the query* $t_2(t_1)$*, namely, it detects paths of length exactly* $(2^2)^2 = 16$*. In general, one can construct an order-1 degree-2 term of the form*

$$t_2^n(t_1) \;=\; \underbrace{t_2\big(\ldots t_2}_{n\ \textit{times}}(t_1)\ldots\big)$$

*that returns the set of all pair of nodes connected by paths of length exactly* $2^{2^n}$ *(i.e.,* 2*, squared* $n$ *times).*

From a standard argument in functional programming (similar results occur in the context of Nested Relational Algebra and functional query languages, see e.g., [31]) one can see that unnormalized terms that use higher-order variables can be arbitrarily more succinct than normalized terms. This is shown, for instance, in the first claim of Proposition 9 below. What is less well-noted, perhaps, is that an additional exponential blowup is required when translating to 'flat' unions of conjunctive queries. In particular, as shown in the second claim of Proposition 9, unnormalized terms of degree 0 (i.e., terms that use only relational variables) are double-exponentially more succinct than unions of conjunctive queries.

Below, we use $\mathsf{tow}_d(n)$ to denote the tower of exponentials that is formally defined by $\mathsf{tow}_0(n) = n$ and $\mathsf{tow}_{d+1}(n) = 2^{\mathsf{tow}_d(n)}$.

**Proposition 9.** *For every* $d \in \mathbb{N}$*, there are terms in* $\mathsf{HO}_1^d[\mathsf{SPJ}^{\mathsf{sing}}]$ *(i.e., using variables of order* $d$ *but evaluating to a query) of size* $\mathcal{O}(n)$ *such that any equivalent expression of Relational Algebra is of size at least* $\mathsf{tow}_{d+1}(n)$*.*
*For every* $d \in \mathbb{N}$*, there are terms in* $\mathsf{HO}_1^d[\mathsf{RA}^+]$ *of size* $\mathcal{O}(n)$ *such that any equivalent union of conjunctive queries is of size at least* $\mathsf{tow}_{d+2}(n)$*.*

The proof of the above proposition is based on the use of typed variants of Church numerals to iterate a large number of times functions such as the one defined by the term $t_1$ of Example 8. The following lemma provides a very general statement for constructing iterations of functional terms, and will be used several times in this paper.

**Lemma 10.** *Let* $k \in \mathbb{N}$ *and let* $t$ *be a term of type* $\tau \to \tau$ *in any signature* $\Sigma$*. For every* $n \in \mathbb{N}$*, there are terms* $t^{\mathsf{tow}_k(n)}$ *in the same signature* $\Sigma$ *that have size* $\mathcal{O}(n)$ *and degree* $\mathsf{max}\big(\mathsf{degree}(t), \mathsf{order}(\tau) + k\big)$*, and that evaluate to the* $\mathsf{tow}_k(n)$*-fold iterations of the function* $[\![t]\!]$*.*

*Proof.* As already mentioned, we will use typed variants of Church numerals. These are terms of the form

$$[n]_\tau \;=\; \lambda \mathtt{Y}.\, \lambda \mathtt{X}.\, \Big( \underbrace{\mathtt{Y}\big(\ldots \mathtt{Y}(\mathtt{X}) \ldots\big)}_{n \text{ occurrences of } Y} \Big)$$

where $\mathtt{Y}$ is a variable of type $\tau \to \tau$ and $\mathtt{X}$ is a variable of type $\tau$. In particular, the Church numeral $[n]_\tau$ has type $(\tau \to \tau) \to \tau \to \tau$ and degree equal to $\mathsf{order}(\tau)+1$. Intuitively, the Church numeral $[n]_\tau$ receives as input a function $f$ of type $\tau \to \tau$ and returns the $n$-fold iteration $f^n$.

We observe that any term of the form $[n]_{\tau\to\tau}\big([2]_\tau\big)$ can be rewritten using $\beta$-reduction into the term $[2^n]_\tau$ – that is, Church numerals of appropriate types can be composed in order to simulate exponentiation. We can iterate this operation to construct terms of degree $\mathsf{order}(\tau) + k + 1$ that succinctly represent

large Church numerals of the form $\big[\mathsf{tow}_k(n)\big]_\tau$:

$$\mathsf{big}^n_{k,\tau} \;=\; \begin{cases} [n]_\tau & \text{if } k = 0, \\ \mathsf{big}^n_{k-1,\tau\to\tau}\big([2]_\tau\big) & \text{if } k > 0. \end{cases}$$

Therefore, for a given number $k$ and a given term $t$ of type $\tau \to \tau$, the term $\mathsf{big}^n_{k,\tau}(t)$ evaluates to the $\mathsf{tow}_k(n)$-fold iteration of the function $[\![t]\!]$. However, it is not yet the desired term, as it has degree $\mathsf{max}\big(\mathsf{degree}(t), \mathsf{order}(\tau)+k+1\big)$. To lower the degree by 1, we note that $\mathsf{big}^n_{k,\tau}(t)$ is a series of redexes of the form

$$\mathsf{big}^n_{k,\tau}(t) \;=\; \Big(\ldots\Big([n]_{\tau_k}\big([2]_{\tau_{k-1}}\big)\Big)\ldots\big([2]_{\tau_0}\big)\Big)\,(t)$$

where $\tau_0 = \tau$ and $\tau_{i+1} = \tau_i \to \tau_i$, and we apply a $\beta$-reduction to the innermost redex, which is either $[n]_{\tau_k}\big([2]_{\tau_{k-1}}\big)$ or $[n]_{\tau_0}(t)$, depending on whether $k > 0$ or $k = 0$. This results in an equivalent term $t^{\mathsf{tow}_k(n)}$ that has size $\mathcal{O}(n)$ and degree $\mathsf{max}\big(\mathsf{degree}(t), \mathsf{order}(\tau)+k\big)$, as desired. □

*Proof of Proposition 9.* We begin by proving the first claim of the proposition. We fix $d \in \mathbb{N}$ and $\tau = (a, b)$, and we recall the definition of the degree-0 term $t_1$ from Example 8:

$$t_1 \;=\; \lambda \mathtt{R}.\; \pi_{a,b}\big(\rho_{b/c}(\mathtt{R}) \bowtie \rho_{a/c}(\mathtt{R})\big).$$

Intuitively, the term $t_1$ receives an input relation $R$ of type $\tau = (a, b)$ and computes the composition $R \circ R$. We can immediately apply Lemma 10 to obtain a family of terms $t_1^{\mathsf{tow}_d(n)}$ of size $\mathcal{O}(n)$ and degree $d$ that compute the $\mathsf{tow}_d(n)$-fold iterations of the query defined by $t_1$. In particular, these terms receive as input a relation $R$ and return the set of all pairs of nodes connected by paths of length exactly $2^{\mathsf{tow}_d(n)} = \mathsf{tow}_{d+1}(n)$.

It remains to show that any expression of Relational Algebra equivalent to $t_1^{\mathsf{tow}_d(n)}$ is bound to have size at least $\mathsf{tow}_{d+1}(n)$. This is done by first observing that Relational Algebra expressions are linearly reducible to formulas of existential first-order logic. Finally, a simple Ehrenfeucht-Fraïsse game argument [19] proves that existential first-order formulas can express existence of paths of length at most linear in the number of quantifiers.

We now turn to the proof of the second claim. Also in this case we iterate $\mathsf{tow}_d(n)$ times a suitable query to detect existence of long paths. To achieve the additional exponential blowup in the size of equivalent Unions of Conjunctive Queries, we nest unions under joins. More precisely, we fix the relational types $\tau = (a, b)$ and $\tau' = (c)$, and the variables $\mathtt{R}, \mathtt{S}, \mathtt{T}$ of types $\tau, \tau', \tau'$, respectively. We then construct the following term with free variables $\mathtt{R}, \mathtt{S}, \mathtt{T}$:

$$u_0 \;=\; \mathtt{R} \;\bowtie\; \rho_{c/a}(\mathtt{S} \cup \mathtt{T}) \;\bowtie\; \rho_{c/b}(\mathtt{S} \cup \mathtt{T})$$

$$u_1 \;=\; \lambda \mathtt{R}.\; \pi_{a,b}\big(\rho_{b/c}(u_0) \bowtie \rho_{a/c}(u_0)\big)$$

In the above definitions the free variable $\mathtt{R}$ is interpreted as a binary relation $R$ that represents a graph, and the free variables $\mathtt{S}, \mathtt{T}$ are interpreted by unary relations $S, T$ that represent subsets of nodes of the graph. Under these assumptions, the term $u_0$ evaluates to the relation $R$ restricted to those pairs of nodes that belong to $S \cup T$. Accordingly, the term $u_1$ defines a query that maps a relation $R$ to the relation $\big(R \cap (S \cup T)^2\big) \circ \big(R \cap (S \cup T)^2\big)$.

Using Lemma 10, we can iterate $\mathsf{tow}_d(n)$ times the query defined by $u_1$, thus obtaining a term of size $\mathcal{O}(n)$ and degree $d$ that receives as input two unary relations $S, T$ and a binary relation $R$, and returns as output the set of all nodes of the graph represented by $R$ that are connected by a path of length $\mathsf{tow}_{d+1}(n)$ consisting only of nodes in $S \cup T$:

$$u_d^n \;=\; \lambda \mathtt{S}.\ \lambda \mathtt{T}.\ u_1^{\mathsf{tow}_d(n)}$$

It remains to show that any union of conjunctive queries equivalent to $u_d^n$ must have size at least $\mathsf{tow}_{d+2}(n)$. Let $\varphi = \varphi_1 \cup \ldots \cup \varphi_N$ be such a union of conjunctive queries. Each disjunct $\varphi_i$ can be thought of as an existential first-order formula of the form $\exists x_{i_1}, \ldots, x_{i_\ell}.\ \alpha_i$, where $\alpha_i$ is a conjunction of atomic predicates over the variables $x_{i_1}, \ldots, x_{i_\ell}$. We consider paths of length $\mathsf{tow}_{d+1}(n)$ whose nodes are labelled over the set $\{\varnothing, \mathtt{S}, \mathtt{T}\}$. We identify any such path $\pi$ with a corresponding triple of relations $(S_\pi, T_\pi, R_\pi)$, where $S_\pi$ (resp., $T_\pi$) consists of all and only the nodes of $\pi$ labelled with $\mathtt{S}$ (resp., $\mathtt{T}$), and $R_\pi$ consists of all and only the edges of $\pi$. Clearly, a path $\pi$ of length $\mathsf{tow}_{d+1}(n)$ satisfies a disjunct of $\varphi$ iff it contains no element labelled with $\varnothing$. It is also easy to see that it could not happen that the same disjunct of $\varphi$ is satisfied by two distinct paths $\pi, \pi'$, that is, paths for which $(S_\pi, T_\pi, R_\pi) \neq (S_{\pi'}, T_{\pi'}, R_{\pi'})$. Otherwise, there would exist a third path $\pi''$ satisfying the same disjunct and containing an element labelled with $\varnothing$. Towards a conclusion, we observe that there exist $2^{\mathsf{tow}_{d+1}(n)}$ distinct paths satisfying $\varphi$. It follows that $\varphi$ contains at least $\mathsf{tow}_{d+2}(n)$ disjuncts. □

## 5. Evaluation of HO terms

This section gives a full picture of the most basic problem concerning terms in higher-order query languages: *evaluation of order-0 terms*, namely, terms that represent database instances. We study this problem not only for higher-order languages based on Relational Algebra, but for *any* collection of relational operators, and also consider the impact of higher-order constants that give greater expressiveness, such as fixpoint operators.

We start with terms of degree 0, whose variables range over database instances. We then extend to the higher-degree case: terms that contain variables of order higher than 0. Here we get tight bounds on the complexity of evaluation by using a technique inspired by Hillebrand and Kannellakis [26], plus observations from Schubert [47] on the complexity of normalization for low-degree terms in the standard $\lambda$-calculus. We will finally consider the impact on evaluation complexity of the meta-query constant $\mathsf{ifp}$, which computes inflationary fixpoints of queries.

The computational complexity of the evaluation problem could be considered for queries defined by order-1 terms and database instances, with respect to the size of the input term (query complexity) or the size of the input relations (data complexity). On the one hand, one easily sees that the data complexity of the evaluation problem collapses to polynomial time for any reasonable signature $\Sigma$. Indeed, for a fixed query term $t$, one can consider the normal form $t^{\downarrow}$ and then reduce to evaluation of fixed queries defined in Non-recursive Datalog, Monad Algebra, Core XQuery, etc. – we know that the data complexity of evaluation of such queries is polynomial time [56, 17, 31]. On the other hand, query complexity turns out to be the same as combined complexity when we consider evaluation problems over signatures that contain all the relational constants and the operators of Relational Algebra (e.g., the signatures $\mathsf{RA}$ and $\mathsf{IFP}$). Indeed, given a query term $t$ and some relations $R_1, \ldots, R_m, S$, we have that

$$[\![t]\!](R_1, \ldots, R_m) = S \qquad \text{iff} \qquad \begin{cases} [\![\pi_{\varnothing}((t\,(R_1)\ \ldots\ t(R_m)) \setminus S)]\!] = \mathsf{false} \\ [\![\pi_{\varnothing}(S \setminus (t\,(R_1)\ \ldots\ t(R_m)))]\!] = \mathsf{false}\ . \end{cases}$$

This means that, as concerns the signatures $\mathsf{RA}$ and $\mathsf{IFP}$, we could equally restrict to *evaluation of terms of boolean type.*

**Definition 11.** *The* evaluation problem *takes as input a closed term $t$ of boolean type over a fixed signature $\Sigma$ and consists of deciding whether $[\![t]\!] = \mathsf{true}$.*

Table 1 summarizes the main complexity results for evaluation of higher-order terms in the signatures $\mathsf{SPJ}$, $\mathsf{RA}^+$, $\mathsf{RA}$, and $\mathsf{IFP}$ (all bounds are tight). Our results imply that the complexity of evaluation is non-elementary for $\mathsf{HO}$ queries with arbitrary high degree, as one might expect from prior results in the $\lambda$-calculus. Since the upper bounds rely only on an analysis of $\beta$-reduction and the complexity of evaluation of the term algebra over the constants, one can easily accommodate other built-in query transformations and database operations.

| **Degree** | **Signatures** $\mathsf{SPJ}$, $\mathsf{RA}^+$, $\mathsf{RA}$ | **Signature** $\mathsf{IFP}$ |
|---|---|---|
| 0 | PSPACE (Prop. 12, 13) | EXP (Prop. 18, 19) |
| $2k-1$ | $k$-EXP (Th. 14) | $k$-EXP (Prop. 18) |
| $2k$ | $k$-EXPSPACE (Th. 14) | $k$-EXPSPACE (Prop. 18) |

Table 1: Complexity of evaluation of $\mathsf{HO}$ terms.

### *5.1. Evaluation of degree-0 terms*

To evaluate a boolean term in $\mathsf{HO}_0^0[\mathsf{RA}]$ one could simply use Proposition 6 and translate it to an equivalent program of Non-recursive Datalog with Stratified Negation. As evaluation of the latter type of programs can be done in polynomial space [56, 29, 17], we immediately get the following result:

**Proposition 12.** *The evaluation problem for $\mathsf{HO}_0^0[\mathsf{RA}]$ is in* PSPACE.

The PSPACE upper bound is tight even for terms with no negation and no union:

**Proposition 13.** *The evaluation problem for* $\mathsf{HO}_0^0[\mathsf{SPJ}]$ *is* PSPACE*-hard.*

*Proof.* We give a reduction from the problem of checking *emptiness of intersection of finite state automata*, which is known to be PSPACE-hard from [33]. This problem consists of deciding, given a tuple of (non-deterministic) finite state automata $\mathcal{A}_1, \ldots, \mathcal{A}_m$ over a common alphabet $\Sigma$, whether there is a string that is accepted by all automata $\mathcal{A}_1, \ldots, \mathcal{A}_m$. Without loss of generality, we can assume that each automaton $\mathcal{A}_i$ has a single initial state and a single final state. Moreover, we observe that adding an $\varepsilon$-labelled self-loop on a state of an automaton does not modify the recognized language. In particular, assuming the presence of such loops on the final states of $\mathcal{A}_1, \ldots, \mathcal{A}_m$ will allow us to consider arbitrary long runs inside automata.

We identify the states of each automaton with integers and we assume that 0 (resp., 1) is the initial (resp., final) state. By identifying in a similar way the letters (including $\varepsilon$) with integers, we can represent the transition function of each automaton $\mathcal{A}_i$ by means of a relation $R_i$ of type $\tau = (s_i, a, t_i)$ such that, for all tuples $\bar{c}$ of type $\tau$, $\bar{c} \in R_i$ iff $\mathcal{A}_i$ contains a $\bar{c}(a)$-labelled transition from state $\bar{c}(s_i)$ to state $\bar{c}(t_i)$. Note that the defined relations $R_1, \ldots, R_m$ share the same attribute name $a$ for the letters consumed by the transitions, while they have pairwise distinct attribute names $s_1, \ldots, s_m$, $t_1, \ldots, t_m$ for the source and target states.

Now, the intersection of $\mathcal{A}_1, \ldots, \mathcal{A}_m$ is obtained from the synchronous product of the relations $R_1, \ldots, R_m$, which is in turn represented by the term

$$t_0 \;=\; \pi_{\bar{s},\bar{t}}\,(R_1 \bowtie \ldots \bowtie R_m)$$

where $\bar{s}$ and $\bar{t}$ are shorthands for the attributes $s_1, \ldots, s_m$ and $t_1, \ldots, t_m$, respectively. Clearly, the intersection language is non-empty iff the relation $[\![t_0]\!]$ contains a path from the initial configuration $(0, \ldots, 0)$ to the final configuration $(1, \ldots, 1)$. Thanks to the presence of self-loops in the final states of $\mathcal{A}_1, \ldots, \mathcal{A}_m$, we can assume that the length of such a path (if it exists) is exactly $n^m$, where $n$ is the maximum number of states in the automata $\mathcal{A}_1, \ldots, \mathcal{A}_m$. Therefore, the existence of a path witnessing non-emptiness of intersection can be detected by iterating $m \cdot (\lfloor \lg n \rfloor + 1)$ times a query $t_1$ similar to that of Example 8. More precisely, we can construct the following terms of degree 0 in the signature $\mathsf{SPJ}$:

$$\begin{aligned}
t_1 \;&=\; \lambda \mathtt{R}.\; \pi_{\bar{s},\bar{t}}\left(\rho_{\bar{t}/\bar{q}}(\mathtt{R}) \bowtie \rho_{\bar{s}/\bar{q}}(\mathtt{R})\right) \\
t_1^*(t_0) \;&=\; \underbrace{t_1\big(\ldots t_1}_{m\cdot(\lfloor \lg n \rfloor + 1)\text{ times}}(t_0)\ldots\big) \\
t_{\mathsf{nonempty}} \;&=\; \pi_{\varnothing}\left(\sigma_{\substack{\bar{s}=0\ldots0 \\ \bar{t}=1\ldots1}}\left(t_1^*(t_0)\right)\right)
\end{aligned}$$

One can easily verify that the term $t_{\mathsf{nonempty}}$ evaluates to $\mathsf{true}$ iff the automata $\mathcal{A}_1, \ldots, \mathcal{A}_m$ accept a common string over $\Sigma$. □

*5.2. Evaluation of higher-degree terms*

We fix for the rest of this section a natural number $k \geq 1$, which defines the degree of the terms we are about to evaluate. We recall that $k$-Exp refers to the class of functions computable in time $\mathsf{tow}_k(n^{\mathcal{O}(1)})$, and similarly for $k$-ExpSpace. Our main result on evaluation is the following:

**Theorem 14.** *The problem of evaluating terms of degree $d \geq 1$ in any signature among $\mathsf{SPJ}, \mathsf{RA}^+, \mathsf{RA}$ is:*

- *$k$-Exp-complete if $d$ is odd, with $d = 2k - 1$,*
- *$k$-ExpSpace-complete if $d$ is even, with $d = 2k$.*

In particular, the evaluation problem for terms of arbitrary high degree has *non-elementary complexity*. We also observe that the complexity of evaluation increases from $k$-Exp (resp., $k$-ExpSpace) to $(k+1)$-Exp (resp., $(k+1)$-ExpSpace) when the degree increases by 2. Note that this is different from the upper bounds that can be obtained by simply applying $\beta$-reduction to terms: with this technique each increase by 1 in the degree would entail an exponential increase in the upper bound.

We begin by proving the upper bounds of Theorem 14. We need however a preliminary upper bound on the time complexity of evaluation of a term $t$ in normal form. We formalize this upper bound as a function of three parameters derived from $t$: the *number $m$ of relational subterms* of $t$, the *maximum arity $h$* of the relational subterms of $t$, and the *total number $\ell$ of active domain elements* (e.g., distinct integers) contained in the relational constants of $t$.

**Lemma 15.** *Let $t$ be a term in $\mathsf{HO}_0^{\downarrow}[\mathsf{RA}]$ with $m$ relational subterms, all of them having arity at most $h$, and with a total number $\ell$ of values. One can evaluate $t$ in time $\mathcal{O}(m \cdot \ell^h)$.*

*Proof.* The idea is to evaluate $t$ in a bottom-up fashion, starting from the relational constants. Since all relational subterms have arity at most $h$, each intermediate result will be a relation of cardinality at most $\ell^h$. Overall, $m$ partial evaluations are performed, each one in time $\mathcal{O}(\ell^h)$. □

*Proof of Theorem 14 (upper bounds).* We prove the claim of the theorem by a complete induction on the degree $d$. Recall that a PSpace upper bound for evaluation of terms of degree $d = 0$ has already been proven in Proposition 12. This serves as the base case of our inductive proof. For the induction step, we assume that $d \geq 1$ and we distinguish two cases depending on whether $d$ is odd or even.

- We first explain how to evaluate a term $t$ of odd degree $d = 2k - 1$, with $k \geq 1$, in time $\mathsf{tow}_k(|t|^{\mathcal{O}(1)})$. For this, we will exploit the fact that we can evaluate any term $r$ of degree $d' = k - 1$ $(< d)$ in time $\mathsf{tow}_k(|r|^{\mathcal{O}(1)})$. Note that for $k = 1$ this follows from Proposition 12 and for $k > 1$ this follows easily from the inductive hypothesis.

Let $r_1, \ldots, r_m$ be the outermost subterms of $t$ having degree at most $k-1$ (and hence order at most $k$). We extend the signature of $t$ with new constants $\rho_1, \ldots, \rho_m$ – each one representing a particular subterm $r_i$ – and we think of $t$ as a new term in the expanded signature. Formally, we define an extension $\iota'$ of the underlying interpretation $\iota$ of $\Sigma$ that maps each constant $\rho_i$ to the evaluation $[\![r_i]\!]_\iota$ of the corresponding subterm $r_i$. We then define $t'$ to be the term obtained from $t$ by replacing every subterm $r_i$ with the constant $\rho_i$. Note that $[\![t]\!]_\iota = [\![t']\!]_{\iota'}$.

We perform standard $\beta$-reduction to compute the normal form $t'^{\downarrow}$ of $t'$: this can be done in time $\mathsf{tow}_k(|t|^{\mathcal{O}(1)})$. We observe that this normalization step may significantly increase the size of the term. However, the number $m'$ of relational subterms in $t'^{\downarrow}$ does not exceed $\mathsf{tow}_k(|t|^{\mathcal{O}(1)})$; moreover, the maximum arity $h$ of these subterms and the total number $\ell$ of values does not change.

We can then evaluate $t'^{\downarrow}$ in a bottom-up fashion, as described in Lemma 15: this can be done in time $\mathcal{O}(m' \cdot \ell^h)$ using an 'oracle' for accessing the semantics of the constants $\rho_1, \ldots, \rho_m$. More precisely, each time we need to extend the partial evaluation to a relational subterm $r$ of $t'^{\downarrow}$ that contains a constant $\rho_i$, we launch a subroutine that evaluates $r$ in time $\mathsf{tow}_k(|r|^{\mathcal{O}(1)})$. Note that this is possible thanks to the inductive hypothesis and the fact that $r$ can be seen as a term of degree at most $k-1$. This shows that $t$ can be evaluated in time $\mathsf{tow}_k(|t|^{\mathcal{O}(1)})$.

- We now consider a term $t$ of even degree $d = 2k$, with $k \geq 1$, and we show how to evaluate it in space $\mathsf{tow}_k(|t|^{\mathcal{O}(1)})$. To do so, we will exploit the inductive hypothesis to evaluate subterms $r$ of $t$ of degree at most $k$ in time $\mathsf{tow}_k(|r|^{\mathcal{O}(1)})$.

  As before, we introduce new constants $\rho_1, \ldots, \rho_m$ representing the outermost subterms $r_1, \ldots, r_m$ of $t$ having degree at most $k$. We accordingly extend the underlying interpretation $\iota$ by letting $\iota'(\rho_i) = [\![r_i]\!]_\iota$, and we let $t'$ to be the term obtained from $t$ by replacing every subterm $r_i$ with $\rho_i$.

  We compute the normal form $t'^{\downarrow}$ of $t'$ and we observe that the number $m'$ of arising relational subterms is at most $\mathsf{tow}_k(|t|^{\mathcal{O}(1)})$. Moreover, we let $h$ be the maximum arity of the relational subterms of $t'$ (or, equally, $t'^{\downarrow}$) and $\ell$ be the total number of values in $t'$(or, equally, $t'^{\downarrow}$).

  We can guess some evaluations for the $m'$ relational subterms of $t'^{\downarrow}$ in space $m' \cdot \ell^h$, which is dominated by $\mathsf{tow}_k(|t|^{\mathcal{O}(1)})$ since $|t'| = \mathsf{tow}_k(|t|^{\mathcal{O}(1)})$, $h \leq |t|$, and $1 \leq k$. Finally, we verify that the guessed evaluations are correct under the fixed semantics of the constants in $t'^{\downarrow}$. Note that the latter step requires accessing the semantics of the constants $\rho_1, \ldots, \rho_m$ according to the extended interpretation $\iota'$: this can be done in time $\mathsf{tow}_k(|t|^{\mathcal{O}(1)})$ thanks to the inductive hypothesis and the fact that the terms $r_1, \ldots, r_m$ have degree at most $k$ ($< d$).

□

We now turn to the lower bounds of Theorem 14, which are obtained via reductions from halting problems of deterministic Turing machines working in time/space $\mathsf{tow}_k(n)$.

We begin by recalling Lemma 10, which gives us a way to succinctly define iterations of functional terms. Specifically, given $n \in \mathbb{N}$ and a term $t$ of a certain type $\tau \to \tau$, one can compute in time $\mathcal{O}(n)$ a term $t^{\mathsf{tow}_k(n)}$ of type $\tau \to \tau$ and degree $\mathsf{max}\big(\mathsf{degree}(t), \mathsf{order}(\tau) + k\big)$ such that

$$[\![t^{\mathsf{tow}_k(n)}]\!] \;=\; [\![t]\!]^{\mathsf{tow}_k(n)}$$

In the following we will make extensively use of terms such as $t^{\mathsf{tow}_k(n)}$ to iterate a large number of times the transition function of a Turing machine, as well as to encode large ordered sets, e.g., tapes of Turing machines. Specifically, we view tapes of Turing machines as data structures similar to doubly linked lists, in which the elements (i.e., the cells) can be accessed in a sequential way. Below, we explain how to scan through the cells of a tape of length $\mathsf{tow}_k(n)$ by means of terms of appropriate degree. For a technical reason – specifically, to avoid that the order of the considered objects becomes too large – we will give definitions in two base cases, $k = 1$ and $k = 2$, and then proceed naturally by induction.

**Definition 16.** *Let $\bar{b} = (b_1, \ldots, b_n)$ be a tuple of attributes with values ranging over $\mathsf{rng}(b_1) = \ldots = \mathsf{rng}(b_n) = \{0, 1\}$.*

- *A $(1,n)$-cell is any singleton relation $S = \{\bar{c}\}$ of type $\tau_{1,n} = (\bar{b})$; we associate with it an* index *between $0$ and $2^n - 1$, defined by $\mathsf{idx}(S) = \sum_{0 \le j < n} 2^j \cdot \bar{c}[j]$.*
- *A $(2,n)$-cell is any relation $R$ of type $\tau_{2,n} = (\bar{b})$; its* index *is a number between $0$ and $2^{2^n} - 1$, defined by $\mathsf{idx}(R) = \sum_{\bar{c}_j \in R} 2^j$, where $\{\bar{c}_0\} < \ldots < \{\bar{c}_{2^n-1}\}$ are all the $(1,n)$-cells ordered according to their indices.*
- *A $(k,n)$-cell, for $k \ge 3$, is any function $F$ of type $\tau_{k,n} = \tau_{k-1,n} \to ()$ mapping $(k-1,n)$-cells to boolean values; its* index *is a number between $0$ and $\mathsf{tow}_k(n) - 1$, defined by $\mathsf{idx}(F) = \sum_{F(C_i)=\mathsf{true}} 2^i$, where $C_0 < \ldots < C_{\mathsf{tow}_{k-1}(n)-1}$ are all the $(k-1,n)$-cells ordered according to their indices.*

In the following we will use $\tau_{k,n}$ to denote the type of a $(k,n)$-cell, as defined above. We remark that the order of this type $\tau_{k,n}$ is $\mathsf{max}(0, k-2)$ and the total number of $(k,n)$-cells is $\mathsf{tow}_k(n)$. Thus, $(k,n)$-cells can be identified with the positions of a tape of a Turing machine working in space $\mathsf{tow}_k(n)$.

Below, we show how to construct suitable terms that scan through all $(k,n)$-cells like in a doubly linked list, where the order on cells is inherited from the natural order of their indices. For the sake of simplicity, in the following we will use the full signature $\mathsf{RA}$ to construct such terms, and we will eventually reduce the acceptance problem of a Turing machine to the evaluation problem for terms in the signature $\mathsf{RA}$. Towards the end of this section we will explain how to exploit a technique from [22, 58] to avoid the use of disjunctions and

negations (i.e., the query constants $\cup$ and $\setminus$), thus obtaining the desired hardness results for the evaluation of terms in the signature SPJ.

**Lemma 17.** *One can efficiently construct the following terms over signature* RA*:*

- $t^{\mathsf{first}}_{k,n}$ *and* $t^{\mathsf{last}}_{k,n}$*, which have type* $\tau_{k,n}$ *and degree* $\mathsf{max}(0, k-3)$*, and evaluate to the first and last* $(k,n)$*-cells, respectively;*
- $t^{=}_{k,n}$*,* $t^{<}_{k,n}$*, and* $t^{>}_{k,n}$*, which have type* $\tau_{k,n} \to \tau_{k,n} \to ()$ *and degree* $2k-2$*, and evaluate to functions that receive two* $(k,n)$*-cells* $C, C'$ *and return* true *iff* $C = C'$ *(resp.,* $C < C'$*,* $C > C'$*);*
- $t^{+1}_{k,n}$ *and* $t^{-1}_{k,n}$*, which have type* $\tau_{k,n} \to \tau_{k,n}$ *and degree* $2k-2$*, and evaluate to the successor and predecessor functions on* $(k,n)$*-cells.*

*Proof.* We start by constructing the terms for the base case $k = 1$. The first $(1,n)$-cell coincides with the singleton relation $\{\bar{0}\}$ of type $\tau_{1,n} = (b_1, \ldots, b_n)$, so we let $t^{\mathsf{first}}_{1,n} = \{\bar{0}\}$. Similarly, the last $(1,n)$-cell is represented by the term $t^{\mathsf{last}}_{1,n} = \{\bar{1}\}$. It is also easy to compare two input $(1,n)$-cells with respect to any relation $\theta$ among $=$, $<$, $>$:

$$t^{\theta}_{1,n} \quad = \quad \lambda \mathtt{S}.\ \lambda \mathtt{S}'.\ \ \pi_{\varnothing}\Big(\sigma_{\theta}\big(\mathtt{S} \ \bowtie \ \rho_{\bar{b}/\bar{b}'}(\mathtt{S}')\big)\Big)$$

where $\bar{b}' = (b'_1, \ldots, b'_n)$ is a fresh copy of the tuple of attributes $\bar{b} = (b_1, \ldots, b_n)$ and, depending on $\theta$ being $=$, $<$, or $>$, the condition for the selection operator $\sigma_{\theta}$ is defined to be either $\bigwedge_{0 \le i < n} \big(a_i = a'_i\big)$, $\bigvee_{0 \le i < n} \big(a_i = 0 \wedge a'_i = 1 \wedge \bigwedge_{j > i} a_i = a'_i\big)$, or $\bigvee_{0 \le i < n} \big(a_i = 1 \wedge a'_i = 0 \wedge \bigwedge_{j > i} a_i = a'_i\big)$. The terms $t^{+1}_{1,n}$ and $t^{-1}_{1,n}$ for the successor and predecessor functions on $(1,n)$-cells are defined in a similar way.

We now turn to constructing the terms that manipulate $(k,n)$-cells, for $k \ge 2$. Recall that the objects that represent $(k,n)$-cells have different types depending on whether $k = 2$ and $k \ge 3$: in the former case, the objects are relations, in the latter case, they are functions. Thus, to define the first and last $(k,n)$-cells, we distinguish between two cases:

- If $k = 2$, then the first $(k,n)$-cell is the empty relation of type $\tau_{2,n} = (b_1, \ldots, b_n)$, so we let $t^{\mathsf{first}}_{2,n} = \varnothing$. Similarly, the last $(k,n)$-cell is the relation that contains all tuples in $\mathsf{rng}(b_1) \times \ldots \times \mathsf{rng}(b_n)$, so we let $t^{\mathsf{last}}_{2,n} = \rho_{b/b_1}(\mathbb{B}) \bowtie \ldots \bowtie \rho_{b/b_n}(\mathbb{B})$, where $\mathbb{B}$ is the relational constant $\{0,1\}$ of type $(b)$.
- If $k \ge 3$, then the first and last $(k,n)$-cells are the constant functions that map all $(k-1,n)$-cells to the boolean values false and true, respectively. These are defined by the terms $t^{\mathsf{first}}_{k,n} = \lambda \mathtt{C}.\ \mathsf{false}$ and $t^{\mathsf{last}}_{k,n} = \lambda \mathtt{C}.\ \mathsf{true}$.

The remaining terms are defined inductively using terms on $(k-1,n)$-cells. In order to simplify the notation, we identify a $(2,n)$-cell. That is, a relation $R$ of type $\tau_{2,n} = (b_1, \ldots, b_n)$, with its characteristic function $F$ that maps any $(1,n)$-cell $\{\bar{c}\}$ to either true or false depending on whether $\bar{c} \in R$ or $\bar{c} \notin R$. Accordingly,

given a $(1,n)$-cell $C = \{\bar{c}\}$, we write $F(C)$ to denote the boolean value $\pi_\emptyset(R \bowtie C)$. These conventions allow us to define in a uniform way terms that manipulate $(2,n)$-cells and terms that manipulate $(k,n)$-cells, for all $k \geq 3$.

Consider two input $(k,n)$-cells $F, F'$ that need to be compared with respect to a relationship among $=$, $<$, $>$. We introduce a new attribute $b$, whose values range over $\mathsf{rng}(b) = \{0, -1, +1\}$. Then, on the basis of $F$ and $F'$, we recursively define a series of functions $G_1, G_2, G_3, \ldots$ of type $\tau_{k-1,n} \to ()$, as follows:

$$G_1(C) = \begin{cases} 0 & \text{if } F(C) \leftrightarrow F'(C) \\ -1 & \text{if } \neg F(C) \wedge F'(C) \\ +1 & \text{if } F(C) \wedge \neg F'(C) \end{cases} \qquad G_{j+1}(C) = \begin{cases} G_j(C-1) & \text{if } G_j(C) = 0 \\ G_j(C) & \text{otherwise} \end{cases}$$

where $C$ denotes a generic $(k-1,n)$-cell (i.e., a formal argument of the functions $F$ and $F'$) and $C-1$ denotes the predecessor of $C$, according to the order on $(k-1,n)$-cells induced by their indices.

It is easy to see that the function $G_{\mathsf{tow}_{k-1}(n)}$ maps the last $(k-1,n)$-cell to either 0, $-1$, or $+1$, depending on whether $F = F'$, $F < F'$, or $F > F'$. We explain now how to implement these tests by means of terms $t^{=}_{k,n}$, $t^{<}_{k,n}$, $t^{>}_{k,n}$ of degree $2k-2$. Let $\mathtt{F}, \mathtt{F'}$ be some variables of type $\tau_{k,n}$ that represent the input $(k,n)$-cells that we want to compare. The initial function $G_1$ is defined by the following term of degree $k-2$:

$$\begin{aligned} t_1 \;=\; \lambda \mathtt{C}.\, \Big( \big(\mathtt{F(C)} \bowtie \mathtt{F'(C)}\big) \;\cup\; \big(\mathsf{true} \setminus \mathtt{F(C)}\big) \bowtie \big(\mathsf{true} \setminus \mathtt{F'(C)}\big) \Big) \;\bowtie\; \{0\} \\ \cup \Big( \big(\mathsf{true} \setminus \mathtt{F(C)}\big) \;\bowtie\; \mathtt{F(C)} \Big) \;\bowtie\; \{-1\} \\ \cup \Big( \mathtt{F(C)} \;\bowtie\; \big(\mathsf{true} \setminus \mathtt{F(C)}\big) \Big) \;\bowtie\; \{+1\}. \end{aligned}$$

Similarly, the transformation from any function $G_j$ to the next function $G_{j+1}$ in the series above can be implemented, in a uniform way, by the following term of type $\big(\tau_{k-1,n} \to ()\big) \to \big(\tau_{k-1,n} \to ()\big)$ and degree $k-1$:

$$\begin{aligned} t_{\mathsf{next}} \;=\; \lambda \mathtt{G}.\, \lambda \mathtt{C}.\, \pi_\emptyset\Big(\sigma_{b=0}\big(\mathtt{G(C)}\big)\Big) \;\bowtie\; \mathtt{G}\big(t^{-1}_{k-1,n}(\mathtt{C})\big) \\ \cup\, \pi_\emptyset\Big(\sigma_{b\neq 0}\big(\mathtt{G(C)}\big)\Big) \;\bowtie\; \mathtt{G}\big(\mathtt{C}\big). \end{aligned}$$

We can use Lemma 10 to compute the $\mathsf{tow}_{k-1}(n)$-fold iteration of the term $t_{\mathsf{next}}$, which results in a term $t^{\mathsf{tow}_{k-1}(n)}_{\mathsf{next}}$ of degree $k-1+k-1 = 2k-2$. We then construct the terms that compare two input $(k,n)$-cells $F, F'$ as follows:

$$\begin{aligned} t^{=}_{k,n} &= \lambda \mathtt{F}.\, \lambda \mathtt{F'}.\;\; \pi_\emptyset\Big( \sigma_{b=0}\big( t^{\mathsf{tow}_{k-1}(n)}_{\mathsf{next}}\; (t_1)\; (t^{\mathsf{last}}_{k-1,n}) \big) \Big) \\ t^{<}_{k,n} &= \lambda \mathtt{F}.\, \lambda \mathtt{F'}.\;\; \pi_\emptyset\Big( \sigma_{b=-1}\big( t^{\mathsf{tow}_{k-1}(n)}_{\mathsf{next}}\; (t_1)\; (t^{\mathsf{last}}_{k-1,n}) \big) \Big) \\ t^{>}_{k,n} &= \lambda \mathtt{F}.\, \lambda \mathtt{F'}.\;\; \pi_\emptyset\Big( \sigma_{b=+1}\big( t^{\mathsf{tow}_{k-1}(n)}_{\mathsf{next}}\; (t_1)\; (t^{\mathsf{last}}_{k-1,n}) \big) \Big). \end{aligned}$$

One can use similar techniques to construct the terms $t^{+1}_{k,n}$ and $t^{-1}_{k,n}$ defining the successor and predecessor functions on $(k,n)$-cells. □

We are now ready to complete the proof of Theorem 14.

*Proof of Theorem 14 (lower bounds).* We begin by reducing the acceptance problem for a deterministic Turing machine running in time $\mathsf{tow}_k(n)$ to an evaluation problem for terms of degree $d = 2k-1$ in the signature $\mathsf{RA}$ – this will imply that the latter problem is $k$-Exp-hard. The deterministic Turing machine is represented as a tuple $\mathcal{M} = (\Sigma, Q, q_0, q_{\mathsf{accept}}, q_{\mathsf{reject}}, \Delta)$, where:

- $\Sigma$ is the tape alphabet, which contains a distinguished "blank" symbol $\square$,
- $Q$ is the finite set of control states,
- $q_0, q_{\mathsf{accept}}, q_{\mathsf{reject}} \in Q$ are, respectively, the initial, accepting, and rejecting states (they are all distinct),
- $\Delta : (Q \setminus \{q_{\mathsf{accept}}, q_{\mathsf{reject}}\}) \times \Sigma \to Q \times \Sigma \times \{-1, +1\}$ is the transition function, which maps the current state and tape symbol to a new state, a new tape symbol, and a direction of movement for the head.

We consider a generic input $w$ of length $n$ and we assume that $\mathcal{M}$ halts after exactly $\mathsf{tow}_k(n) - 1$ transitions. The input word is either accepted or rejected depending on the halting state being $q_{\mathsf{accept}}$ or $q_{\mathsf{reject}}$. In the following, we show how to construct, in time polynomial in $n$, a term $t_{\mathsf{accept}}$ of degree $d = 2k-1$ that evaluates to $\mathsf{true}$ iff $\mathcal{M}$ accepts the input $w$.

Observe that $\mathcal{M}$ visits at most $\mathsf{tow}_k(n)$ cells of its tape, and hence these cells can be put in bijection with the $(k,n)$-cells defined above. We can then represent the content of the tape of $\mathcal{M}$ by a function mapping $(k,n)$-cells to letters. More precisely, we introduce two new attributes whose values range over $\Sigma$ and $Q \uplus \{\bot\}$; for the sake of simplicity, we let $\Sigma$ and $Q_\bot$ be the names of these new attributes. We encode a configuration of $\mathcal{M}$ by a function $F$ of type $\tau_{k,n} \to (\Sigma, Q_\bot)$, which maps a $(k,n)$-cell $C$ to a singleton relation of the form $\{(a_C, q_C)\}$, where the first value $a_C$ describes the *symbol* contained in the cell $C$, while the second value $q_C$ describes the *current state* of $\mathcal{M}$, provided that the head lies exactly on the cell $C$, otherwise $q_C = \bot$.

We recall that, for all $k \geq 2$, the order of a $(k,n)$-cell is $k-2$, and hence the order of the function that encodes a configuration of $\mathcal{M}$ is $k-1$. Unfortunately, this is not true for $k = 1$. Because the previous property is important for generating terms of the correct degree, we need to modify the definition of encoding of a configuration when $k = 1$: in this case, the encoding is given by a relation $R$ of type $(\bar{b}, \Sigma, Q_\bot)$ that contains *exactly one tuple* $(\bar{c}, a, q)$ for each $(1,n)$-code $\{\bar{c}\}$. By a slight abuse of notation, we will still identify the encoding $R$ with the function $F$ that maps any $(1,n)$-code $C = \{\bar{c}\}$ to the singleton $\{(a,q)\}$ such that $(\bar{c}, a, q) \in R$, and accordingly write $F(C)$ in place of $\pi_{\Sigma,Q_\bot}(R \bowtie C)$.

Below we construct, on the basis of the (fixed) Turing machine $\mathcal{M}$ and the input word $w = a_1 \dots a_n$, a series of terms of size polynomial in $n$:

- The term $t_0$ represents the initial configuration of $\mathcal{M}$. To construct this term we need to distinguish between the first cell of the tape, where the input symbol $a_1$ appears and the head of $\mathcal{M}$ lies with its initial state $q_0$, the next $n-1$ cells, which contain the letters $a_2, \ldots, a_n$, and the remaining cells that contain the blank symbol. The term $t_0$ can be constructed using the terms $t^{\mathsf{first}}_{k,n}$, $t^{=}_{k,n}$, $t^{+1}_{k,n}$, $t^{>}_{k,n}$ available from Lemma 17, and it turns out to have degree $2k-2$:

$$\begin{aligned} t_0 \;=\; & \lambda \mathtt{C}.\; t^{=}_{k,n}\big(\mathtt{C}\big)\big(t^{\mathsf{first}}_{k,n}\big) \;\bowtie\; \big\{(a_1, q_0)\big\} \\ & \cup \bigcup_{1 \le i < n} t^{=}_{k,n}\big(\mathtt{C}\big)\Big(\underbrace{t^{+1}_{k,n}\big(\ldots t^{+1}_{k,n}\big(}_{i \text{ times}}\; t^{\mathsf{first}}_{k,n}\big)\ldots\big)\Big) \;\bowtie\; \big\{(a_i, \bot)\big\} \\ & \cup\; t^{>}_{k,n}\big(\mathtt{C}\big)\Big(\underbrace{t^{+1}_{k,n}\big(\ldots t^{+1}_{k,n}\big(}_{n-1 \text{ times}}\; t^{\mathsf{first}}_{k,n}\big)\ldots\big)\Big) \;\bowtie\; \big\{(\square, \bot)\big\} \end{aligned}$$

  (the above definition can be easily modified in the case $k = 1$ so as to define a relation of type $(\bar{b}, a, q)$ instead of a function of type $(\bar{b}) \to (a, q)$).

- For each transition $\delta = (q, a, q', a', d) \in Q \times \Sigma \times Q \times \Sigma \times \{-1, +1\}$ of $\mathcal{M}$, we construct the following terms of type $\big(\tau_{k,n} \to (\Sigma, Q_\bot)\big) \to (\Sigma, Q_\bot)$:

$$t^{\mathsf{exit}}_{\delta} \;=\; \lambda \mathtt{F}.\; \lambda \mathtt{C}.\;\; \pi_{\varnothing}\Big(\sigma_{=(a,q)}\big(\mathtt{F}(\mathtt{C})\big)\Big) \;\bowtie\; \big\{(a', \bot)\big\}$$

$$t^{\mathsf{enter}}_{\delta} \;=\; \lambda \mathtt{F}.\; \lambda \mathtt{C}.\;\; \pi_{\varnothing}\Big(\sigma_{=(a,q)}\big(\mathtt{F}\big(t^{-d}_{k,n}(\mathtt{C})\big)\big)\Big) \;\bowtie\; \pi_{\Sigma}\big(\mathtt{F}(\mathtt{C})\big) \;\bowtie\; \big\{q'\big\}$$

$$t^{\mathsf{other}}_{\delta} \;=\; \lambda \mathtt{F}.\; \lambda \mathtt{C}.\;\; \pi_{\varnothing}\Big(\sigma_{\neq(a,q)}\big(\mathtt{F}(\mathtt{C})\big)\Big) \bowtie \pi_{\varnothing}\Big(\sigma_{\neq(a,q)}\big(\mathtt{F}\big(t^{-d}_{k,n}(\mathtt{C})\big)\big)\Big) \;\bowtie\; \mathtt{F}(\mathtt{C}).$$

  Intuitively, the term $t^{\mathsf{exit}}_{\delta}$ receives the current configuration of $\mathcal{M}$ and a tape cell $C$ and, if the head is on $C$ and the transition $\delta$ is enabled, it returns the updated content of the cell $C$; otherwise, it returns the empty relation. Similarly, the term $t^{\mathsf{enter}}_{\delta}$ receives the current configuration of $\mathcal{M}$ and a tape cell $C$ and, if $C$ is the cell reached by the head after executing the transition $\delta$, it returns the content of the cell $C$ with the updated current state; otherwise, it returns the empty relation. The term $t^{\mathsf{other}}_{\delta}$ returns the content of all remaining cells, which are not affected by the specified transition. Note that the above terms have degree $2k-2$.

- Using the above terms, we compute the effect of a single transition of $\mathcal{M}$:

$$t_{\mathsf{step}} \;=\; \lambda \mathtt{F}.\; \lambda \mathtt{C}.\;\; \bigcup_{\delta \in \Delta}\Big(t^{\mathsf{exit}}_{\delta}\big(\mathtt{F}(\mathtt{C})\big) \cup t^{\mathsf{enter}}_{\delta}\big(\mathtt{F}(\mathtt{C})\big)\Big) \;\cup\; \bigbowtie_{\delta \in \Delta}\Big(t^{\mathsf{other}}_{\delta}\big(\mathtt{F}(\mathtt{C})\big)\Big).$$

  Note that, for all $k \ge 2$, the order of the formal argument $\mathtt{F}$ in the above term is $\mathsf{order}\big(\tau_{k,n} \to (\Sigma, Q_\bot)\big) = k-1$. Similarly, for $k = 1$, the variable $\mathtt{F}$ has relational type $(\bar{b}, \Sigma, Q_\bot)$ and hence degree $0 = k-1$.

- Using Lemma 10, we compute the $\mathsf{tow}_k(n)$-fold iteration of the term $t_{\mathsf{step}}$: this results in a term $t^{\mathsf{tow}_k(n)}_{\mathsf{step}}$ of degree $\mathsf{max}(2k-2,\, k-1+k) = 2k-1$.

- Finally, we define $t_{\mathsf{accept}} = \pi_{\varnothing}\big(\sigma_{Q_\perp = q_{\mathsf{accept}}}\big(t_{\mathsf{step}}^{\mathsf{tow}_k(n)}(t_0)\big)\big)$.

We have just shown that the evaluation problem for terms of degree $d = 2k-1$ in the signature RA is $k$-Exp-hard. A very similar proof shows that the evaluation problem for terms of degree $d = 2k$ is $k$-ExpSpace-hard: for this it is sufficient to consider the computation of a deterministic Turing machine $\mathcal{M}$ on a tape of length $\mathsf{tow}_k(n)$ (the machine thus halts within $\mathsf{tow}_{k+1}(n)$ steps), and accordingly iterate $\mathsf{tow}_{k+1}(n)$ times the term $t_{\mathsf{step}}$ defined above.

To conclude the proof we need to explain how to avoid, both in Lemma 17 and in the above reduction, the use of the query constants $\cup$ and $\setminus$, so as to obtain analogous hardness results for the evaluation problem of terms in the signature SPJ. The solution is based on the standard method of encoding the membership (resp., non-membership) of any tuple by means of an additional attribute with value 1 (resp., 0) – a method presented in [22, 58]. More precisely, we introduce a new attribute $b$ of domain $\mathsf{rng}(b) = \{0,1\}$, and we replace every relational constant $R$ of type $\tau$ with a new constant $\tilde{R}$ of type $(\tau, b)$, defined by

$$\tilde{R} \;=\; \big\{(\bar{c},1) \;:\; \bar{c} \in R\big\} \;\cup\; \big\{(\bar{c},0) \;:\; \bar{c} \in \mathsf{rng}(\tau) \setminus R\big\}.$$

Of course, the above definition makes sense only when the domain $\mathsf{rng}(\tau)$ is finite, which is the case for all the relational constants that appear in the terms defined so far. Thanks to the above replacement, one can simulate union and set difference by means of joins. More precisely, one introduces two relational constants And, Or of type $(b_1, b_2, b)$, representing the truth tables of conjunction and disjunction, and a third table Neg of type $(b_3, b_2)$, representing the truth table of negation. Then one rewrites all (sub)terms with a relational type of the form $(\bar{a}, b)$ by applying the following rules:

$$\begin{aligned}
t_1 \;\bowtie\; t_2 \;&\to\; \pi_{\bar{a},b}\big(\,\rho_{b/b_1}(t_1') \;\bowtie\; \mathsf{And} \;\bowtie\; \rho_{b/b_2}(t_2')\,\big)\\
t_1 \;\cup\; t_2 \;&\to\; \pi_{\bar{a},b}\big(\,\rho_{b/b_1}(t_1') \;\bowtie\; \mathsf{Or} \;\bowtie\; \rho_{b/b_2}(t_2')\,\big)\\
t_1 \;\setminus\; t_2 \;&\to\; \pi_{\bar{a},b}\big(\,\rho_{b/b_1}(t_1') \;\bowtie\; \mathsf{And} \;\bowtie\; \mathsf{Neg} \;\bowtie\; \rho_{b/b_3}(t_2')\,\big).
\end{aligned}$$

□

*5.3. Adding recursion*

Here we study the impact on evaluation complexity of adding the second-order constant ifp to the signature. The proposition below shows that terms in the signature IFP with degree higher than 1 can be rewritten into equivalent terms in the signature RA and with the same degree.

**Proposition 18.** *Every term of* $\mathsf{HO}_0^d[\mathsf{IFP}]$ *can be efficiently rewritten into an equivalent term of* $\mathsf{HO}_0^{\mathsf{max}(d,1)}[\mathsf{RA}]$.

*Proof.* Let $t$ be a term of $\mathsf{HO}_0^d[\mathsf{IFP}]$ and let $n$ be its size. Recall that $n$ is at least the size of the relational constants in $t$. Since the size of any intermediate relation $R$ defined by a subterm of $t$ is bounded by $\mathcal{O}(2^n)$, we can calculate

the inflationary fixpoint $\mathsf{ifp}(Q, R)$, for any query $Q$, as $\tilde{Q}^{2^n}(R)$, where $\tilde{Q}$ is the query defined by $\lambda \mathtt{R}.\ Q(\mathtt{R}) \cup \mathtt{R}$. Lemma 10 implies that the $2^n$-fold iteration of the query $\tilde{Q}$ can be defined by a suitable term of degree $\mathsf{max}(d, 1)$. Moreover the latter term can be efficiently computed from the formal argument $Q$ of the $\mathsf{ifp}$ operator. In particular, we can efficiently transform any subterm of $t$ of the form $\mathsf{ifp}(t_1, t_2)$ into an equivalent term $\big(\lambda \mathtt{R}.\ t_1(\mathtt{R}) \cup \mathtt{R}\big)^{\mathsf{tow}_1(n)}(t_2)$ of degree $\mathsf{max}\big(\mathsf{degree}(t_1), \mathsf{degree}(t_2), 1\big)$. Doing this iteratively gives the desired transformation. □

As a consequence of the above proposition, we have that the complexity of evaluating terms of degree strictly higher than 0 does not change when we add the constant $\mathsf{ifp}$ to the signature. On the other hand, we see that the $\mathsf{ifp}$ constant does have some impact on the complexity of degree-0 evaluation:

**Proposition 19.** *The evaluation problem for* $\mathsf{HO}^0_0[\mathsf{IFP}]$ *is* Exp*-complete.*

The above upper bound is inherited from the case of terms of degree 1. Hardness follows from the Exp-hardness of Recursive Datalog in query complexity [17], and the fact that Recursive Datalog can be easily embedded in $\mathsf{HO}^0_0[\mathsf{IFP}]$ (cf. Proposition 6).

## 6. Containment of HO terms

In this section we study a variant of the containment problem for HO terms. This will be based on a notion of containment for higher-order queries which we will define just below.

For terms of order 0, the definition of containment is straightforward: given two closed terms $t, t'$ of the same relational type $\tau$ and given an interpretation $\iota$ for the common signature, we write

$$t \subseteq_\iota t' \qquad \text{iff} \qquad [\![t]\!]_\iota \subseteq [\![t']\!]_\iota.$$

We extend the definition of the containment relation from relational terms to higher-order terms as follows. Let $t$ and $t'$ be two closed terms that have $\beta$-normal forms $\lambda \mathtt{X}_1 \ldots \lambda \mathtt{X}_n.q$ and $\lambda \mathtt{X}_1 \ldots \lambda \mathtt{X}_n.q'$, where $q$ and $q'$ use no abstraction, and have the same type $\tau_1 \to \ldots \to \tau_n \to \tau$, where $\tau$ is a relational type. In addition, let $\iota$ be an interpretation for the common signature of the terms $t, t'$. We write

$$t \subseteq_\iota t' \qquad \text{iff} \qquad \forall 1 \le i \le n.\ \forall \mathrm{O}_i \in \mathsf{dom}(\tau_i).\ [\![q]\!]_{\iota[\mathtt{X}_i \mapsto \mathrm{O}_i]} \subseteq [\![q']\!]_{\iota[\mathtt{X}_i \mapsto \mathrm{O}_i]}.$$

Intuitively, the containment relation for higher-order terms is defined in a pointwise manner by considering the values of the denoted functions on all possible inputs. As usual, we will assume that the underlying interpretation $\iota$ is the standard one for the considered signature and we will accordingly omit the subscript $\iota$ from the containment notation.

**Example 20.** *Let* $\mathtt{R}$ *be a variable of relational type* $\tau$ *and let* $\mathtt{Q}$ *be a variable of query type* $\tau \to \tau$*. Consider the following terms over the common signature* $\mathsf{IFP}$*:*

$$\begin{aligned} t &= \lambda\mathtt{Q}.\ \lambda\mathtt{R}.\ \pi_{\varnothing}\big(\,\mathtt{Q}(\mathtt{R})\,\big) \\ t' &= \lambda\mathtt{Q}.\ \lambda\mathtt{R}.\ \pi_{\varnothing}\big(\,\mathsf{ifp}(\mathtt{Q})(\mathtt{R})\,\big). \end{aligned}$$

*Given the standard interpretation of the order-*2 *constant* $\mathsf{ifp}$ *(cf. Section 4), we have that* $\mathsf{ifp}(Q)(R) = \bigcup_{i\in\mathbb{N}} Q^i(R)$*, and hence* $Q(R) \subseteq \mathsf{ifp}(Q)(R)$ *for all queries* $Q$ *of type* $\tau \to \tau$ *and all relations* $R$ *of type* $\tau$*. This shows that* $t \subseteq t'$*.*

We will also compare higher-order terms *on subsets* of their domains. For example, in Section 6.1 we will compare meta-queries – i.e., functions mapping queries to relations – by restricting their inputs to be those queries that are definable in the Positive Relational Algebra. We denote by $\mathsf{dom}(\mathtt{X}_i)$ the ranges of the formal arguments $\mathtt{X}_i$ that are relevant for the containment relation, and we collectively call these ranges *the base* of the containment relation. For simplicity, we will only consider 'uniform' bases for query variables, that is, bases that associate with all formal arguments of order 2 the sets of queries of appropriate types that can be constructed from a unique common signature. We will specify the base of the containment relation as a superscript of the symbol $\subseteq$, using a suggestive notation. For example, given two terms $t, t'$ of order 2 in the signature $\mathsf{RA}$, we may write $t \subseteq^{\lambda\mathsf{SPJ}} t'$ to denote the fact that, when the input queries for both $t$ and $t'$ are restricted to range over queries definable in $\mathsf{HO}_1^{\downarrow}[\mathsf{SPJ}]$, then the function denoted by $t$ is point-wise contained in the function denoted by $t'$.

In the following, we give a couple of examples that show how order-2 containment may depend on the underlying base.

**Example 21.** *Let* $\mathtt{R}, \mathtt{S}$ *be two variables of relational type* $\tau = (a)$*, with* $\mathsf{rng}(a) = \mathbb{Z}$*, and* $\mathtt{Q}$ *a variable of query type* $\tau \to ()$*. Consider the two terms in* $\mathsf{HO}_2^{\downarrow}[\mathsf{RA}^+]$*:*

$$\begin{aligned} t &= \lambda\mathtt{Q}.\ \lambda\mathtt{R}.\ \lambda\mathtt{S}.\ \ \mathtt{Q}(\mathtt{R}) \\ t' &= \lambda\mathtt{Q}.\ \lambda\mathtt{R}.\ \lambda\mathtt{S}.\ \ \mathtt{Q}(\mathtt{R} \cup \mathtt{S})\ . \end{aligned}$$

*For all relations* $R, S$ *and all monotone queries* $Q$ *– in particular, for all queries of Positive Relational Algebra – we have* $Q(R) \subseteq Q(R \cup S)$*. This shows that* $t \subseteq^{\lambda\mathsf{RA}^+} t'$*.*

*On the other hand, if the base allows us to instantiate the variable* $\mathtt{Q}$ *with queries of full Relational Algebra, then we can choose* $R = \{1\}$*,* $S = \{2\}$*, and* $Q = [\![\lambda\mathtt{T}.\ \mathsf{true} \setminus \pi_{\varnothing}(\sigma_{a=2}(\mathtt{T}))]\!]$ *as instances of the variables* $\mathtt{R}$*,* $\mathtt{S}$*, and* $\mathtt{Q}$*, respectively, in such a way that* $Q(R) = \mathsf{true} \not\subseteq \mathsf{false} = Q(R \cup S)$*. This shows that* $t \not\subseteq^{\lambda\mathsf{RA}} t'$*.*

**Example 22.** *Let* $\mathtt{R}$ *be again a variable of relational type* $\tau = (a)$ *and let* $\mathtt{Q}$ *be a variable of query type* $\tau \to \tau$*. Consider the following terms in* $\mathsf{HO}_2^{\downarrow}[\mathsf{RA}^+]$*:*

$$\begin{aligned} t &= \lambda\mathtt{Q}.\ \lambda\mathtt{R}.\ \mathtt{Q}\big(\mathtt{Q}(\mathtt{R})\big) \\ t' &= \lambda\mathtt{Q}.\ \lambda\mathtt{R}.\ \mathtt{Q}(\mathtt{R})\ . \end{aligned}$$

*We have* $t \not\subseteq^{\lambda\mathsf{SPJ}} t'$*. Indeed, we can instantiate the variable* $\mathtt{Q}$ *by the query* $Q = [\![\lambda \mathtt{T}.\ \pi_b\big(\mathtt{T} \bowtie \{(1,2),(2,3)\}\big)]\!]$*, where* $\{(1,2),(2,3)\}$ *is a relational constant of type* $(a,b)$*. In this way we get* $Q(\{1\}) = \{2\}$ *and* $Q(\{2\}) = \{3\}$*, and hence* $[\![t]\!](Q)(\{1\}) = \{3\} \not\subseteq \{2\} = [\![t']\!](Q)(\{1\})$*.*

*On the other hand, we can prove that* $t \subseteq^{\lambda\mathsf{SPJ}^{\mathsf{sing}}} t'$*. This follows essentially from the fact that any query* $Q$ *of Relational Algebra that witnesses a non-containment of term* $t$ *in term* $t'$ *must use either a union or a non-singleton relational constant, which are both forbidden in the signature* $\mathsf{SPJ}^{\mathsf{sing}}$*. Indeed, consider a query* $Q$ *definable in the signature* $\mathsf{SPJ}^{\mathsf{sing}}$ *and an arbitrary relation* $R$ *as instances of the formal parameters of the terms* $t$ *and* $t'$*. Recall that* $Q$ *is equivalent to a conjunctive query that receives the unary relation* $R$ *and outputs another unary relation. In particular,* $Q$ *can be defined by one of the following rules:*

$$Q(c)\ :-\qquad\qquad Q(c)\ :-\ R(c)\qquad\qquad Q(x)\ :-\ R(x)$$

*where* $c$ *is a single-value constant, e.g., the integer* $1$*, and* $x$ *is a variable. A simple case distinction based on which of the above rules defines* $Q$*, shows that* $Q$ *is idempotent, that is* $Q\big(Q(R)\big) = Q(R)$*, and hence* $t \subseteq^{\lambda\mathsf{SPJ}^{\mathsf{sing}}} t'$*.*

The rest of the section is devoted to analyse the complexity of the containment problem for $\mathsf{HO}$ terms, formally defined as follows:

**Definition 23.** *Let* $\mathcal{C}, \mathcal{C}'$ *denote two classes of* $\mathsf{HO}$ *terms over a common signature* $\Sigma$ *and let* $\Lambda$ *be a base. The* containment problem *with base* $\Lambda$ *for left-hand side terms in* $\mathcal{C}$ *and right-hand side terms in* $\mathcal{C}'$ *consists of deciding, given two terms* $t \in \mathcal{C}$ *and* $t' \in \mathcal{C}'$ *of the same type, whether* $t \subseteq^{\Lambda} t'$*.*

It is worth remarking that the containment problem subsumes several crucial problems related to (higher-order) queries and, more generally, functional programs, such as *satisfiability* – given a term $t$, decide whether $[\![t]\!](\bar{O}) = \mathsf{true}$ for some input object $\bar{O}$ – and *extensional equivalence* – given two terms $t, t'$, decide whether $[\![t]\!](\bar{O}) = [\![t']\!](\bar{O})$ for all possible input objects $\bar{O}$. Accordingly, two terms $t, t'$ are extensionally equivalent iff they satisfy the two containments $t \subseteq t'$ and $t' \subseteq t$.

We will always consider the computational complexity of the containment problem with respect to the *size* of the terms, as defined earlier in Section 4. We will study the complexity of containment in two different cases: when terms are normalized, and when terms are unnormalized. In the case of normalized terms, the complexity of containment is fairly independent of the syntax of the calculus, depending rather on the base for the containment relation (e.g., the range of the query variables). For instance, in Corollary 38 we will show that the containment problem for normalized terms in $\mathsf{HO}^{\downarrow}_2[\mathsf{RA}^+]$ with monotone queries as base is $\Pi^{\mathrm{P}}_2$-complete, and thus has the same complexity as containment between simple expressions of Positive Relational Algebra [46]. Additionally, we will give complexity results in the presence of integrity constraints. In the case of unnormalized terms, the problem has an additional source of complexity, related to

the phenomenon of *sharing* of subterms during $\beta$-reductions – it is exactly the source of complexity that is eliminated in considering normalized terms. We will only consider the containment problem for unnormalized terms of order 1, which evaluate to ordinary queries. Since these terms take as input tuples of arbitrary relations, we can also omit the base in the notation of containment.

Table 2 summarizes the main complexity results, classified by order and degree of the terms. We annotate these results with the references or statements where they are actually proved; all the complexity upper bounds in the table are provably tight, except for the last two in the right column that follows from the complexity of the containment problem for order-2 normalized terms via a series of $\beta$-reductions.

| **Order** | **Degree** | **Normalized terms** | **Unnormalized terms** |
|---|---|---|---|
| 1 | 0 | $\Pi_2^{\mathrm{P}}$ [46] | coNExp (Prop. 24) |
| 1 | $d \geq 1$ | undefined | co-$(d{+}1)$-NExp (Th. 27) |
| 2 | 1 | $\Pi_2^{\mathrm{P}}$ (Th. 28) | $\Pi_2^{2\text{-Exp}}$ (Cor. 38) |
| 2 | $d \geq 2$ | undefined | $\Pi_2^{(d+1)\text{-Exp}}$ (Cor. 38) |

Table 2: Complexity of containment of HO terms with signature and base $\mathsf{RA}^+$.

### *6.1. Containment of order-1 terms*

We examine here the containment problem for terms of order 1, that is, terms that evaluate to queries. We start with terms containing variables of low order, and then extend the results to terms of higher degrees.

**Low-degree terms.** Here we analyse the complexity of the containment problem for terms of order 1 and degree 0. Since satisfiability of boolean expressions of full Relational Algebra is undecidable, we will mainly focus only on terms in the signature $\mathsf{RA}^+$. Moreover, because terms of degree 0 in the signature $\mathsf{RA}^+$ correspond to Non-recursive Datalog programs, the complexity results that follow can be seen as simple implications of a series of results known in the literature. We report the main arguments only briefly.

**Proposition 24.** *The containment problem between left-hand side and right-hand side terms in* $\mathsf{HO}_1^0[\mathsf{RA}^+]$ *is* coNExp*-complete, and it is* coNExp*-hard even when the right-hand side terms are normalized and in the signature* $\mathsf{SPJ}^{\mathsf{sing}}$.

*Proof.* By Proposition 6, containment between terms in $\mathsf{HO}_1^0[\mathsf{RA}^+]$ is the same as containment of Non-recursive Datalog programs. Theorems 3 and 7 from [6] give the desired complexity bounds. □

The complexity of containment decreases when we restrict the class of left-hand side terms. Specifically, we show that the containment problem becomes PSpace-complete when left-hand side terms are in the signature $\mathsf{SPJ}^{\mathsf{sing}}$:

**Proposition 25.** *The containment problem between left-hand side terms in* $\mathsf{HO}_1^0[\mathsf{SPJ}^{\mathsf{sing}}]$ *and right-hand side terms in* $\mathsf{HO}_1^0[\mathsf{RA}^+]$ *is* PSPACE*-complete, and it is* PSPACE*-hard even when the right-hand side terms are in* $\mathsf{HO}_1^0[\mathsf{SPJ}]$.

*Proof.* The PSPACE upper bound follows essentially from Proposition 12 in [6], which shows that the problem of determining whether a Conjunctive Non-recursive Datalog program (i.e., a Non-recursive Datalog program in which every intentional predicate occurs on the left-hand side of at most one rule) is contained in another Non-recursive Datalog program is in PSPACE. The link between Datalog programs and query terms of degree 0 is provided by Proposition 6: thanks to this correspondence, any left-hand side term in $\mathsf{HO}_1^0[\mathsf{SPJ}^{\mathsf{sing}}]$ can be translated into an equivalent Conjunctive Non-recursive Datalog program, and similarly any right-hand side term in $\mathsf{HO}_1^0[\mathsf{RA}^+]$ can be translated into an equivalent Non-recursive Datalog program.

The hardness result follows from a reduction from the evaluation problem for terms in $\mathsf{HO}_0^0[\mathsf{SPJ}]$, which was shown to be PSPACE-hard in Proposition 13. More precisely, given a boolean term $t \in \mathsf{HO}_0^0[\mathsf{SPJ}]$, one reduces the problem of deciding whether $t$ evaluates to $\mathsf{true}$ to the containment $\lambda\mathtt{Q}.\,\mathsf{true} \subseteq \lambda\mathtt{Q}.\,t$, where $\mathtt{Q}$ is a dummy query variable used to construct terms of order 1. □

By further restricting the right-hand side terms to be in the signature $\mathsf{SPJ}^{\mathsf{sing}}$ and in normal form, one gets a better bound for the containment problem:

**Proposition 26.** *The containment problem between left-hand side terms in* $\mathsf{HO}_1^0[\mathsf{SPJ}^{\mathsf{sing}}]$ *and right-hand side terms in* $\mathsf{HO}_1^{\downarrow}[\mathsf{SPJ}^{\mathsf{sing}}]$ *is NP-complete.*

*Proof.* A proof of this result, using the terminology for Datalog programs, is presented in Proposition 13 in [6]. □

**Higher-degree terms.** We now turn to considering terms in the signature $\mathsf{RA}^+$ having degree higher than 0. The following theorem shows that each increase in the degree corresponds to jumping two levels higher in the complexity hierarchy.

**Theorem 27.** *For every* $d \geq 1$*, the containment problem between terms in* $\mathsf{HO}_1^d[\mathsf{RA}^+]$ *is* CO-$(d+1)$-NEXP*-complete.*

*Proof.* The complexity upper bound follows directly from Proposition 24 and the fact that every term $t$ of degree $d$ can be transformed into an equivalent term $t'$ of degree 0 with size blow-up $\mathsf{tow}_d\big(\mathcal{O}(|t|)\big)$.

Below, we prove the hardness result by exploiting a reduction from the non-acceptance problem for a non-deterministic Turing Machine that runs in time $\mathsf{tow}_{d+1}(n)$ on inputs of length $n$. The non-deterministic Turing machine is represented by a tuple $\mathcal{M} = (\Sigma, Q, q_0, q_{\mathsf{accept}}, q_{\mathsf{reject}}, \Delta)$, where $\Sigma$, $Q$, $q_0$,$q_{\mathsf{accept}}$, $q_{\mathsf{reject}}$ are as in the proof of Theorem 14, and $\Delta : \big((Q \setminus \{q_{\mathsf{accept}}, q_{\mathsf{reject}}\}) \times \Sigma\big) \times \big(Q \times \Sigma \times \{-1, +1\}\big)$ is a transition relation describing the possible target configurations on the basis of the current configuration.

We consider a generic input $w$ of length $n$ and we assume that $\mathcal{M}$ halts after exactly $\mathsf{tow}_{d+1}(n) - 1$ transitions. From $\mathcal{M}$ and $w$, we will construct two terms $t, t' \in \mathsf{HO}^d_1[\mathsf{RA}^+]$ such that

$$\mathcal{M} \text{ accepts } w \qquad \text{iff} \qquad t \not\subseteq t'.$$

More precisely, the terms $t, t'$ will denote queries mapping relations to boolean values. In particular, $t \not\subseteq t'$ will hold iff there exist two relations $R, S$ such that $[\![t]\!](R)(S) = \mathsf{true}$ and $[\![t']\!](R)(S) = \mathsf{false}$. If this happens, the first relation $R$ will encode a successful run of $\mathcal{M}$ on $w$, while the second relation $S$ will encode a list of elements to be used to compare the configurations of $\mathcal{M}$ at consecutive computation steps, and the content of a tape at consecutive positions. We remark that the terms $t$ and $t'$ must avoid the use of the set difference operator, since they must be in the signature $\mathsf{RA}^+$. For this reason, the properties that characterize the relations $R, S$ will be divided into positive and negative ones; positive properties will be captured by the left-hand side term $t$, while negative properties will be captured, in negated form, by the right-hand side term $t'$. Before constructing these terms, we explain how we encode a run of $\mathcal{M}$ on $w$.

Recall that a run of $\mathcal{M}$ on any input of length $n$ consists of exactly $\mathsf{tow}_{d+1}(n)$ configurations, and each configuration consists of at most $\mathsf{tow}_{d+1}(n)$ tape positions with non-blank symbols. We can put in bijection each computation step (resp., each tape position) of $\mathcal{M}$ with an integer $i$ (resp., $j$) ranging over an ordered domain of size at least $\mathsf{tow}_{d+1}(n)$. Accordingly, a run of $\mathcal{M}$ can be described by a relation $R$ consisting of exactly one tuple $(i, j, a_{i,j}, q_{i,j})$ for each pair $(i, j)$, where the value $a_{i,j}$ describes the tape symbol at computation step $i$ and at tape position $j$, and the value $q_{i,j}$ describes the current state of $\mathcal{M}$ at computation step $i$, provided that the head lies at position $j$, otherwise $q_{i,j} = \bot$. Hereafter, we let $I, J, \Sigma, Q_\bot$ be some attribute names with ranges $\mathsf{rng}(I) = \mathsf{rng}(J) = \mathbb{Z}$, $\mathsf{rng}(\Sigma) = \Sigma$, and $\mathsf{rng}(Q_\bot) = Q \uplus \{\bot\}$. We then let $\mathtt{R}$ be a variable of relational type $(I, J, \Sigma, Q_\bot)$ that represents a candidate run of $\mathcal{M}$.

In order to be able to reason on consecutive computation steps of $\mathcal{M}$ and consecutive tape positions, we use a list consisting of $\mathsf{tow}_{d+1}(n)$ distinct integers. Specifically, we first introduce a variable $\mathtt{S}$ of type $(a, b)$, with $\mathsf{rng}(a) = \mathsf{rng}(b) = \mathbb{Z}$, whose instance represents a generic directed graph. We then determine the pairs of nodes in the graph represented by $\mathtt{S}$ that are connected by paths of length at most $\mathsf{tow}_{d+1}(n)$:

$$\mathtt{S}_{\mathsf{connected}} \;=\; \Big(\lambda \mathtt{S}'.\;\; \mathtt{S}' \;\cup\; \pi_{a,b}\big(\,\rho_{b/c}(\mathtt{S}') \bowtie \rho_{a/c}(\mathtt{S}')\,\big)\Big)^{\mathsf{tow}_d(n)}\;\big(\,\mathtt{S}\,\big)$$

(note that the above term is obtained from a $\mathsf{tow}_d(n)$-fold iteration, as described in Lemma 10, and has degree $d$). Next, we restrict to a sub-graph of $\mathtt{S}$ that contains only nodes that are accessible from a distinguished element, say the integer 0, via paths of length at most $\mathsf{tow}_{d+1}(n)$:

$$\mathtt{S}_0 \;=\; \mathtt{S} \;\bowtie\; \pi_b\Big(\,\sigma_{a=0}\big(\,\mathtt{S}_{\mathsf{connected}}\,\big)\Big)$$

It remains to enforce the condition that $\mathtt{S}_0$ represents a graph in which every node has *in-degree at most* 1 and *out-degree at most* 1 – note that this condition implies that the graph consists of a single path with no cycle, that is, a list. Enforcing this condition amounts at verifying that that there exist no pairs $(x, y)$ and $(x', y')$ in the relation defined by $\mathtt{S}_0$ such that $x = x' \wedge y \neq y'$ or $x \neq x' \wedge y = y'$. The negation of the latter property can be defined by a term of degree $d$ in the signature of Positive Relational Algebra (the variable $\mathtt{X}$ below represents a singleton relation $\{x\}$ of type $(a)$):

$$q = \lambda \mathtt{X}.\ \pi_{\varnothing}\Big(\sigma_{b \neq b'}\big(\mathtt{X} \bowtie \mathtt{S}_0 \bowtie \rho_{b/b'}(\mathtt{S}_0)\big) \ \cup\ \sigma_{a \neq a'}\big(\mathtt{S}_0 \bowtie \rho_{a/a'}(\mathtt{S}_0) \bowtie \rho_{a/b}(\mathtt{X})\big)\Big)$$

$$t_{\mathsf{not\text{-}a\text{-}list}} = \Big(\lambda \mathtt{X}.\ q\,(\mathtt{X}) \ \cup\ q\big(\rho_{b/a}(\pi_b(\mathtt{X} \bowtie \mathtt{S}_0))\big)\Big)^{\mathsf{tow}_d(n)}\ \big(\{0\}\big)$$

Summing up, for any instance of the variable $\mathtt{S}$, we have that $[\![t_{\mathsf{not\text{-}a\text{-}list}}]\!] = \mathsf{false}$ iff $\mathtt{S}_0$ represents a list of at most $\mathsf{tow}_{d+1}(n)$ distinct integers starting with 0. In this case we can use the term $\mathtt{S}_0$ to compare configurations of $\mathcal{M}$ at consecutive time points, as well as the content of a tape at consecutive positions.

We are now ready to construct the terms $t$ and $t'$. The first constraint that we need to enforce is the functional dependency $(I, J) \to (\Sigma, Q_\perp)$ on the relational variable $\mathtt{R}$, which represents a candidate run of $\mathcal{M}$. This is done by adding to the right-hand side term $t'$ a disjunct of the form

$$t_{\mathsf{not\text{-}a\text{-}func}} = \bigcup_{\substack{0 \leq i < \mathsf{tow}_{d+1}(n) \\ 0 \leq j < \mathsf{tow}_{d+1}(n)}} \ \bigcup_{(a,q) \neq (a',q')} \pi_{\varnothing}\big(\sigma_{=(i,j,a,q)}(\mathtt{R})\big) \ \bowtie\ \pi_{\varnothing}\big(\sigma_{=(i,j,a',q')}(\mathtt{R})\big)$$

(by a slight abuse of notation, we use aggregate functions over the parameters $0 \leq i, j < \mathsf{tow}_{d+1}(n)$ – assuming that $\mathtt{S}$ encodes a list of $\mathsf{tow}_{d+1}(n)$ elements, the semantics of the above aggregate functions can be succinctly captured by $\mathsf{tow}_{d+1}(n)$-fold iterations of query terms containing the variables $\mathtt{R}$ and $\mathtt{S}$).

Next, we describe the initial configuration of $\mathcal{M}$ on input $w = a_0 \dots a_{n-1}$. This is given by the set of all and only the tuples $(i, j, a_{i,j}, q_{i,j})$, with $i = 0$, that could appear in the interpretation of $\mathtt{R}$. Specifically, we have $a_{0,j} = a_j$ for all $0 \leq j < n$, $a_{0,j} = \square$ for all $n \leq j < \mathsf{tow}_{d+1}(n)$, $q_{0,0} = q_0$, and $q_{0,j} = \perp$ for all $1 \leq j < \mathsf{tow}_{d+1}(n)$. These constraints are captured by adding to the left-hand side term $t$ a conjunct of the form:

$$t_{\mathsf{init}} = \pi_{\varnothing}\big(\sigma_{=(0,0,a_0,q_0)}(\mathtt{R})\big) \ \bowtie \bigbowtie_{1 \leq j < n-1} \pi_{\varnothing}\big(\sigma_{=(0,j,a_j,\perp)}(\mathtt{R})\big) \ \bowtie \bigbowtie_{n \leq j < \mathsf{tow}_{d+1}(n)} \pi_{\varnothing}\big(\sigma_{=(0,j,\square,\perp)}(\mathtt{R})\big).$$

Similarly, termination with acceptance is captured by adding another conjunct to $t$:

$$t_{\mathsf{accept}} = \bigcup_{\substack{0 \leq i < \mathsf{tow}_{d+1}(n) \\ 0 \leq j < \mathsf{tow}_{d+1}(n)}} \ \bigcup_{a \in \Sigma} \pi_{\varnothing}\big(\sigma_{=(i,j,a,q_{\mathsf{accept}})}(\mathtt{R})\big).$$

Finally, we show how to detect violations (for instance, of the transition relation) in the run encoded by $\mathtt{R}$. For short, we denote by $i + 1$ (resp., $j + 1$) the

successor of a computation step $i$ (resp., a tape position $j$). Recall that the successors $i+1$ and $j+1$ can be defined using the variable $\mathtt{S}$ under the assumption that $[\![t_{\mathsf{not\text{-}a\text{-}list}}]\!] = \mathsf{false}$. By a slight abuse of notation, we further use the successors $i+1$ and $j+1$ as arguments of selection operators, and we freely use aggregate functions (all these constructions can be defined by succinct terms). The possible violations in the encoding $\mathtt{R}$ of a run can be witnessed by a disjunct of $t'$ of the following form:

$$
\begin{aligned}
t_{\mathsf{not\text{-}a\text{-}run}} \;=\; & \bigcup_{q,q' \in Q} \pi_{\varnothing}\Big( \sigma_{j \neq j'} \big( \mathtt{R} \;\bowtie\; \rho_{j/j',a/a',q/q'}\,(\mathtt{R}) \big) \Big) \\
\cup\; & \bigcup_{q,q' \in Q} \pi_{\varnothing}\Big( \sigma_{i'=i+1 \,\wedge\, j' \neq j+1 \,\wedge\, j' \neq j-1} \big( \mathtt{R} \;\bowtie\; \rho_{i/i',\,j/j',\,a/a',\,q/q'}\,(\mathtt{R}) \big) \Big) \\
\cup\; & \bigcup_{\substack{0 \leq i < \mathsf{tow}_{d+1}(n) \\ 0 \leq j < \mathsf{tow}_{d+1}(n)}} \; \bigcup_{\substack{a \neq a' \\ q' \in Q \cup \{\bot\}}} \pi_{\varnothing}\big( \sigma_{=(i,j,a,\bot)}(\mathtt{R}) \big) \;\bowtie\; \pi_{\varnothing}\big( \sigma_{=(i+1,j,a',q')}(\mathtt{R}) \big) \\
\cup\; & \bigcup_{\substack{0 \leq i < \mathsf{tow}_{d+1}(n) \\ 0 \leq j < \mathsf{tow}_{d+1}(n)}} \; \bigcup_{\substack{(q,a,q',a',d) \notin \Delta \\ a'' \in \Sigma}} \left( \begin{array}{l} \pi_{\varnothing}\big( \sigma_{=(i,j,a,q)}(\mathtt{R}) \big) \;\bowtie \\ \pi_{\varnothing}\big( \sigma_{=(i+1,j,a',\bot)}(\mathtt{R}) \big) \;\bowtie \\ \pi_{\varnothing}\big( \sigma_{=(i+1,j+d,a'',q')}(\mathtt{R}) \big) \end{array} \right).
\end{aligned}
$$

Putting all together, we define the following terms of degree $d$:

$$
\begin{aligned}
t \;&=\; \lambda \mathtt{R}.\; \lambda \mathtt{S}.\;\; t_{\mathsf{init}} \;\bowtie\; t_{\mathsf{accept}} \\
t' \;&=\; \lambda \mathtt{R}.\; \lambda \mathtt{S}.\;\; t_{\mathsf{not\text{-}a\text{-}list}} \;\cup\; t_{\mathsf{not\text{-}a\text{-}func}} \;\cup\; t_{\mathsf{not\text{-}a\text{-}run}}.
\end{aligned}
$$

One can easily verify that $\mathcal{M}$ accepts $w$ iff there exist some relations $R$ and $S$ such that $[\![t]\!](R)(S) = \mathsf{true}$ and $[\![t']\!](R)(S) = \mathsf{false}$, that is, iff $t \not\subseteq t'$. □

*6.2. Containment of order-2 terms*

The goal of this section is to prove tight bounds on the complexity of the containment problem for *order-2 terms in normal form*, where the signature and the base are fragments of the Relational Algebra. Specifically, we will analyse the complexity of deciding containments of the form $t \subseteq^{\Lambda} t'$, where $t$ and $t'$ are terms in $\mathsf{HO}_2^{\downarrow}[\mathsf{RA}^+]$ and where $\Lambda$ specifies the ranges of the query variables used as formal arguments (cf. beginning of Section 6).

**Base $\mathsf{RA}^+$.** We start by considering the case where the query variables are interpreted by terms of the Positive Relational Algebra.

**Theorem 28.** *The order-2 containment problem with signature $\mathsf{RA}^+$ and base $\lambda\mathsf{RA}^+$, namely, the problem of deciding whether $t \subseteq^{\lambda\mathsf{RA}^+} t'$ for terms $t, t' \in \mathsf{HO}_2^{\downarrow}[\mathsf{RA}^+]$, is $\Pi_2^{\mathrm{P}}$-complete.*

The above theorem is proved by reducing the order-2 containment problem to some variants of containment problems between order-1 terms. More precisely, the containment problems we reduce to are between conjunctive queries (i.e., order-1 terms in the signature $\mathsf{SPJ}^{\mathsf{sing}}$ and in normal form) and queries of Positive Relational Algebra (i.e., terms in the signature $\mathsf{RA}^+$ and in normal form), where the input relations are restricted so as to satisfy sets of constraints of the form $R_i \subseteq R_j$ (*positive constraints*) or $R_i \not\subseteq R_j$ (*negative constraints*). Moreover, we consider disjunctive variants of these constrained containment problems.

**Definition 29.** *Let* $t_1, \ldots, t_n \in \mathsf{HO}_1^{\downarrow}[\mathsf{SPJ}^{\mathsf{sing}}]$ *and* $t'_1, \ldots, t'_n \in \mathsf{HO}_1^{\downarrow}[\mathsf{RA}^{+}]$ *be terms such that, for all* $1 \leq i \leq n$, $t_i$ *and* $t'_i$ *have the same type* $(\tau_1 \times \ldots \times \tau_m) \to \sigma_i$, *for some relational types* $\tau_1, \ldots, \tau_m$, $\sigma_1, \ldots, \sigma_n$. *Further let* $R_1, \ldots, R_m$ *be relational symbols of types* $\tau_1, \ldots, \tau_m$, *respectively, and let* $\Gamma$ *be a set of constraints of the form* $R_i \subseteq R_j$ *or* $R_i \not\subseteq R_j$, *where* $R_i$ *and* $R_j$ *have the same type* $\tau_i = \tau_j$.
*The* constrained disjunctive containment problem *consists of deciding whether, for every instance* $\bar{R} = (R_1, \ldots, R_m)$ *satisfying the constraints in* $\Gamma$, *there exists an index* $1 \leq i \leq n$ *such that* $[\![t_i]\!](\bar{R}) \subseteq [\![t'_i]\!](\bar{R})$. *When this holds, we denote it for short by*

$$\bigvee_{1 \leq i \leq n} t_i \subseteq_{\Gamma} t'_i \ .$$

If the set $\Gamma$ of constraints in the above definition is empty (or it always evaluates to `true`), then the problem is called *unconstrained* disjunctive containment problem and it is denoted by $\bigvee_{1 \leq i \leq n} t_i \subseteq t'_i$. Similarly, if the number $n$ in the above definition is 1, then we simply talk of a (constrained or unconstrained) containment problem.

The first ingredient of the proof of Theorem 28 is the following lemma, which basically shows that the negative constraints in a disjunctive containment problem can be translated to additional disjuncts, and that disjunctions of containments are not more difficult than a single containment.

**Lemma 30.** *The constrained disjunctive containment problem for left-hand side terms in* $\mathsf{HO}_1^{\downarrow}[\mathsf{SPJ}^{\mathsf{sing}}]$ *and right-hand side terms in* $\mathsf{HO}_1^{\downarrow}[\mathsf{RA}^{+}]$ *reduces in polynomial time to the (disjunction-free) constrained containment problem for boolean left-hand side terms in* $\mathsf{HO}_1^{\downarrow}[\mathsf{SPJ}^{\mathsf{sing}}]$ *and boolean right-hand side terms in* $\mathsf{HO}_1^{\downarrow}[\mathsf{RA}^{+}]$, *where no negative containments constraints occur.*

*Proof.* Consider an instance of the constrained disjunctive containment that is given by some terms $t_1, \ldots, t_n \in \mathsf{HO}_1^{\downarrow}[\mathsf{SPJ}^{\mathsf{sing}}]$ and $t'_1, ..., t'_n \in \mathsf{HO}_1^{\downarrow}[\mathsf{RA}^{+}]$, and by a set $\Gamma$ of positive and negative containment constraints. For the sake of simplicity, we assume that all terms have the same type $\bar{\tau} \to \sigma$, where $\bar{\tau} = (\tau_1 \times \ldots \times \tau_m)$ and $\tau_1, \ldots, \tau_m, \sigma$ are relational types. We transform the instance of the disjunctive containment problem to an equivalent instance of the containment problem that uses only positive constraints and terms defining boolean queries. For this, we need to 'hide' the outputs of the queries $t_1, t'_1, \ldots, t_n, t'_n$ under an expanded input structure. We thus introduce new relational types as follows. We denote by $\bar{\sigma} = (\sigma_1, \ldots, \sigma_n)$ the $n$-fold cartesian product of the relational type $\sigma$ with itself, which is defined by taking fresh copies of the attributes in $\sigma$. We then partition the set $\Gamma$ of containment constraints into two subsets, $\Gamma_+$ and $\Gamma_-$, that contain the positive and the negative containment constraints, respectively. We enumerate the negative containment constraints as follows: $\Gamma_- = \left\{R_{i_1} \not\subseteq R_{j_1}, \ \ldots, \ R_{i_\ell} \not\subseteq R_{j_\ell}\right\}$, where $\ell = |\Gamma_-|$ and $1 \leq i_k, j_k \leq n$ for all $1 \leq k \leq \ell$. Note that the presence of a constraint of the form $R_{i_k} \not\subseteq R_{j_k}$ witnesses the fact that the types $\tau_{i_k}$ and $\tau_{j_k}$ coincide; we denote such a type by $\theta_k$. Finally, we define the relational type $\bar{\theta} = \theta_1 \times \ldots \times \theta_\ell$.

We are now ready to explain the reduction. We begin by transforming the terms $t_1, t'_1, \ldots, t_n, t'_n$ into equivalent boolean terms $\bar{t}_1, \bar{t}'_1, \ldots, \bar{t}_n, \bar{t}'_n$ of type $(\bar{\tau} \times \bar{\sigma} \times \bar{\theta}) \to ()$:

$$\bar{t}_i \;=\; \lambda\bar{\mathtt{R}}.\ \lambda\bar{\mathtt{S}}.\ \lambda\bar{\mathtt{T}}.\ \pi_{\varnothing}\big(t_i(\bar{\mathtt{R}}) \cap \mathtt{S}_i\big)$$

$$\bar{t}'_i \;=\; \lambda\bar{\mathtt{R}}.\ \lambda\bar{\mathtt{S}}.\ \lambda\bar{\mathtt{T}}.\ \pi_{\varnothing}\big(t'_i(\bar{\mathtt{R}}) \cap \mathtt{S}_i\big)$$

where $\bar{\mathtt{R}} = (\mathtt{R}_1, \ldots, \mathtt{R}_n)$ ranges over objects of type $\bar{\tau}$, $\bar{\mathtt{S}} = (\mathtt{S}_1, \ldots, \mathtt{S}_m)$ ranges over objects of type $\bar{\sigma}$, and $\bar{\mathtt{T}} = (\mathtt{T}_1, \ldots, \mathtt{T}_\ell)$ ranges over objects of type $\bar{\theta}$ (note that the queries defined by $\bar{t}_i$ and $\bar{t}'_i$ do not depend on the argument $\bar{\mathtt{T}}$, which will be used later).

Given the above definition, we have that $\bar{t}_i(\bar{R}, \bar{S}, \bar{T})$ evaluates to $\mathsf{true}$ iff the relation $S_i$ has non-empty intersection with $[\![t_i]\!](\bar{R})$. Clearly, the analogous property holds for term $t'_i$. Hence we have $[\![t_i]\!](\bar{R}) \subseteq [\![t'_i]\!](\bar{R})$ iff, for all relations $\bar{S}$ of type $\bar{\sigma}$ and all relations $\bar{T}$ of type $\bar{\theta}$, $[\![\bar{t}_i]\!](\bar{R}, \bar{S}, \bar{T}) = \mathsf{true}$ implies $[\![\bar{t}'_i]\!](\bar{R}, \bar{S}, \bar{T}) = \mathsf{true}$ (we write for short $[\![\bar{t}_i]\!](\bar{R}, \bar{S}, \bar{T}) \to [\![\bar{t}'_i]\!](\bar{R}, \bar{S}, \bar{T})$). From this property, we obtain:

$$\forall\ \bar{R} \vDash \Gamma \qquad \bigvee_{1 \le i \le n} \big([\![t_i]\!](\bar{R}) \subseteq [\![t'_i]\!](\bar{R})\big)$$

$$\text{iff} \quad \forall\ \bar{R} \vDash \Gamma \qquad \bigvee_{1 \le i \le n} \Big(\forall\ \bar{S} : \bar{\sigma}\ \ \forall\ \bar{T} : \bar{\theta}\ \ \big([\![\bar{t}_i]\!](\bar{R}, \bar{S}, \bar{T}) \to [\![\bar{t}'_i]\!](\bar{R}, \bar{S}, \bar{T})\big)\Big)$$

$$\text{iff} \quad \begin{array}{l}\forall\ \bar{R} \vDash \Gamma \\ \forall\ \bar{S} : \bar{\sigma}\ \ \forall\ \bar{T} : \bar{\theta}\end{array} \ \Big(\bigwedge_{1 \le i \le n} [\![\bar{t}_i]\!](\bar{R}, \bar{S}, \bar{T})\Big) \to \Big(\bigvee_{1 \le i \le n} [\![\bar{t}'_i]\!](\bar{R}, \bar{S}, \bar{T})\Big).$$

Finally, we show how to get rid of the negative constraints in $\Gamma$ by translating them into appropriate query containment relations. We recall the enumeration $\{R_{i_1} \notin R_{j_1}, \ldots, R_{i_\ell} \notin Rj_\ell\}$ of $\Gamma_-$ and, for every index $1 \le k \le \ell$, we introduce two boolean queries $u_k, u'_k$ of type $(\bar{\tau} \times \bar{\sigma} \times \bar{\theta}) \to ()$:

$$u_k \;=\; \lambda\bar{\mathtt{R}}.\ \lambda\bar{\mathtt{S}}.\ \lambda\bar{\mathtt{T}}.\ \pi_{\varnothing}\big(\mathtt{R}_{i_k} \cap \mathtt{T}_k\big)$$

$$u'_k \;=\; \lambda\bar{\mathtt{R}}.\ \lambda\bar{\mathtt{S}}.\ \lambda\bar{\mathtt{T}}.\ \pi_{\varnothing}\big(\mathtt{R}_{j_k} \cap \mathtt{T}_k\big)$$

(note that the above queries do not depend on the argument $\bar{\mathtt{S}}$). We then consider a generic object $\bar{R} = (R_1, \ldots, R_n)$ of type $\bar{\tau}$. We have that $\bar{R}$ *violates* the negative constraint $R_{i_k} \notin R_{j_k}$ iff, for all relations $\bar{S}$ of type $\bar{\sigma}$ and all relations $\bar{T}$ of type $\bar{\theta}$, $[\![u_k]\!](\bar{R}, \bar{S}, \bar{T}) = \mathsf{true}$ implies $[\![u'_k]\!](\bar{R}, \bar{S}, \bar{T}) = \mathsf{true}$. It follows that

$$\forall\ \bar{R} \vDash \Gamma\ \ \forall\ \bar{S} : \bar{\sigma}\ \ \forall\ \bar{T} : \bar{\theta}$$
$$\Big(\bigwedge_{1 \le i \le n} [\![\bar{t}_i]\!](\bar{R}, \bar{S}, \bar{T})\Big) \to \Big(\bigvee_{1 \le i \le n} [\![\bar{t}'_i]\!](\bar{R}, \bar{S}, \bar{T})\Big)$$

$$\text{iff}\ \ \forall\ \bar{R} \vDash \Gamma_+\ \ \forall\ \bar{S} : \bar{\sigma}\ \ \forall\ \bar{T} : \bar{\theta}$$
$$\Big(\bigwedge_{1 \le i \le n} [\![\bar{t}_i]\!](\bar{R}, \bar{S}, \bar{T}) \wedge \bigwedge_{1 \le k \le \ell} [\![u_k]\!](\bar{R}, \bar{S}, \bar{T})\Big) \to \Big(\bigvee_{1 \le i \le n} [\![\bar{t}'_i]\!](\bar{R}, \bar{S}, \bar{T}) \vee \bigvee_{1 \le k \le \ell} [\![u'_k]\!](\bar{R}, \bar{S}, \bar{T})\Big).$$

The latter property can be viewed as an instance of the containment problem for left-hand side terms in $\mathsf{HO}_1^{\downarrow}[\mathsf{SPJ}^{\mathsf{sing}}]$ and right-hand side terms in $\mathsf{HO}_1^{\downarrow}[\mathsf{RA}^{+}]$, under positive containment constraints. □

We will also need to introduce a transformation of any query of Positive Relational Algebra into an equivalent union of conjunctive queries. Such a transformation, which may imply an exponential blowup, is achieved by 'pushing upward' all occurrences of the union operator. Formally, the transformation rules are as follows:

$$\begin{aligned}
\rho_{\{a/b\}}(t_1 \cup t_2) &\rightsquigarrow \rho_{\{a/b\}}(t_1) \cup \rho_{\{a/b\}}(t_2) \\
\sigma_c(t_1 \cup t_2) &\rightsquigarrow \sigma_c(t_1) \cup \sigma_c(t_2) \\
\pi_A(t_1 \cup t_2) &\rightsquigarrow \pi_A(t_1) \cup \pi_A(t_2) \\
(t_1 \cup t_2) \bowtie t_3 &\rightsquigarrow (t_1 \bowtie t_3) \cup (t_2 \bowtie t_3) \\
t_1 \bowtie (t_2 \cup t_3) &\rightsquigarrow (t_1 \bowtie t_2) \cup (t_1 \bowtie t_3).
\end{aligned}$$

By repeatedly applying the above rules starting from a term $t \in \mathsf{HO}_1^{\downarrow}[\mathsf{RA}^+]$, one obtains a semantically equivalent term of the form

$$\tilde{t} \;=\; \lambda\bar{\mathtt{R}}.\; \tilde{t}_1(\bar{\mathtt{R}}) \;\cup\; \ldots \;\cup\; \tilde{t}_N(\bar{\mathtt{R}})$$

where each subterm $\tilde{t}_i$ is in the signature $\mathsf{SPJ}^{\mathsf{sing}}$, is in normal form, and has size at most $|t|$, and where $N$ is a number bounded by an exponential in the size of $t$. We call the term $\tilde{t}$ the *flattening of* $t$ and we say that each subterm $\tilde{t}_i$ is is a *disjunct* of $\tilde{t}$.

The following lemma shows that the problem of checking whether a given term appears as a disjunct in the flattening of another term is in NP.

**Lemma 31.** *The problem of deciding, given some terms* $t \in \mathsf{HO}_1^{\downarrow}[\mathsf{RA}^+]$ *and* $t' \in \mathsf{HO}_1^{\downarrow}[\mathsf{SPJ}^{\mathsf{sing}}]$*, whether* $t'$ *is a disjunct of the flattening* $\tilde{t}$ *of* $t$ *is in* NP*.*

*Proof.* We identify terms with their syntactic trees. By construction, every disjunct $\tilde{t}_i$ of $\tilde{t}$ can be obtained by removing from the syntactic tree of $t$ all $\cup$-labelled nodes and exactly one of the two subtrees issued from each of these nodes. This gives a simple non-deterministic polynomial-time algorithm that checks whether $t'$ is a disjunct of $\tilde{t}$. □

By putting together Lemma 30 and Lemma 31, we can show that constrained disjunctive containment problem is NP-complete:

**Proposition 32.** *The constrained disjunctive containment problem for left-hand side terms in* $\mathsf{HO}_1^{\downarrow}[\mathsf{SPJ}^{\mathsf{sing}}]$ *and right-hand side terms in* $\mathsf{HO}_1^{\downarrow}[\mathsf{RA}^+]$ *is* NP*-complete.*

*Proof.* NP-hardness holds trivially since the considered problem already includes all instances of the (unconstrained disjunction-free) containment problem for conjunctive queries, which is known to be NP-hard [1]. As for the complexity upper bound, in view of Lemma 30 it is sufficient to prove NP membership of the disjunction-free containment problem under positive constraints only. We can further restrict ourselves to terms defining boolean queries.

As a preliminary step, we recall the notion of *canonical model* of a boolean conjunctive query $Q$ of type $(\tau_1 \times \ldots \times \tau_m) \to ()$. This is defined as the tuple $\bar{R}^Q = (R_1^Q, \ldots, R_m^Q)$, where each $R_i^Q$ is a relation of type $\tau_i$ consisting of all and only the records $\bar{c} = (c_1, \ldots, c_{|\tau_i|})$ such that the predicate $R_i(\bar{c})$ appears in the rule-based definition of $Q$ (note that the predicate $R_i(\bar{c})$ may contain constants and variables, and both are seen as values when constructing the relation $R_i^Q$). We recall the well-known characterization of Chandra and Merlin [13]:

$$\begin{array}{l} \textit{For every tuple } \bar{R} \textit{ of relations, } Q(\bar{R}) = \mathsf{true} \textit{ iff} \\ \textit{there exists a homomorphism from } \bar{R}^Q \textit{ to } \bar{R}. \end{array} \tag{1}$$

The above property will help us in reducing the constrained containment problem to a query evaluation problem.

Let us consider a generic instance of the constrained containment problem, which consists of terms $t \in \mathsf{HO}_1^{\downarrow}[\mathsf{SPJ}^{\mathsf{sing}}]$ and $t' \in \mathsf{HO}_1^{\downarrow}[\mathsf{RA}^+]$, both of type $(\tau_1 \times \ldots \times \tau_m) \to ()$, and a set $\Gamma$ of positive containment constraints over relations $\bar{R}$ of type $\bar{\tau} = \tau_1 \times \ldots \times \tau_m$. Let $Q$ be the conjunctive query defined by $t$. We transform $Q$ into a new query $Q^*$ by applying a variant of the chase procedure for inclusion dependencies [1]. Formally, $Q^*$ is obtained by repeatedly expanding the right-hand side of $Q$ with new predicates of the form $R_j(\bar{c})$, where $\bar{c}$ is a tuple associating variables or constants with the attributes of $\tau_j$, whenever $R_i \subseteq R_j$ is a constraint in $\Gamma$ and the predicate $R_i(\bar{c})$ appears already in the current expansion of the right-hand side. Further let $R_i^{Q^*}$ be the canonical model of the chased conjunctive query $Q^*$. Note that both $Q^*$ and $\bar{R}^{Q^*}$ can be computed in polynomial-time from $t$.

We are now able to prove a reduction from the constrained containment problem to a query evaluation problem:

$$t \subseteq_\Gamma t' \qquad \text{iff} \qquad [\![t']\!](\bar{R}^{Q^*}) = \mathsf{true}.$$

$\Rightarrow$) Assume that $t \subseteq_\Gamma t'$. By definition, the canonical model $\bar{R}^{Q^*}$ satisfies $Q^*$, and hence $Q$ as well. By construction, $\bar{R}^{Q^*}$ also satisfies every containment constraint in $\Gamma$. Thus, knowing that $t \subseteq_\Gamma t'$, we immediately derive $[\![t']\!](\bar{R}^{Q^*}) = \mathsf{true}$.

$\Leftarrow$) Let $Q' = [\![t']\!]$ and assume that $Q(\bar{R}^{Q^*}) = \mathsf{true}$. From (1), we know that there is a homomorphism $h'$ from the canonical model $\bar{R}^{Q'}$ of $Q'$ to the canonical model $\bar{R}^{Q^*}$ of $Q^*$. Consider a generic tuple $\bar{R}$ of relations that satisfies both the query $Q$ and the containment constraints in $\Gamma$. Clearly, by definition of $Q^*$, we have $Q^*(\bar{R}) = \mathsf{true}$. Again from (1), we know that there is another homomorphism $h$ from $\bar{R}^{Q^*}$ to $\bar{R}$. The functional composition $h \circ h'$ is a homomorphism from $\bar{R}^{Q'}$ to $\bar{R}$. By applying once again (1), we conclude that $Q'(\bar{R}) = \mathsf{true}$.

The above reduction yields a non-deterministic polynomial-time algorithm that, given $t \in \mathsf{HO}_1^{\downarrow}[\mathsf{SPJ}^{\mathsf{sing}}]$ and $t' \in \mathsf{HO}_1^{\downarrow}[\mathsf{RA}^+]$, decides whether $t \subseteq_\Gamma t'$. Indeed, it is sufficient to exploit Lemma 31 to non-deterministically guess (i) a disjunct $\tilde{t}'_i$ of the flattening $\tilde{t}'$ of $t'$ and (ii) a homomorphism from $\bar{R}^{Q'_i}$ to $\bar{R}^{Q^*}$, where $Q'_i = [\![\tilde{t}'_i]\!]$: this would witness $[\![t']\!](\bar{R}^{Q^*}) = \mathsf{true}$, and hence $t \subseteq_\Gamma t'$. □

It is convenient now to generalize the containment from relations to tuples of relations: given $\bar{R} = (R_1, \ldots, R_m)$ and $\bar{R}' = (R'_1, \ldots, R'_m)$, we write $\bar{R} \subseteq \bar{R}'$ whenever $R_i \subseteq R'_i$ holds for all indices $1 \leq i \leq m$. Hereafter, we say that a query $Q$ is *monotone* iff, for every tuple $\bar{R} = (R_1, \ldots, R_m)$ and $\bar{R}' = (R'_1, \ldots, R'_m)$ of relations of appropriate types, $\bar{R} \subseteq \bar{R}'$ implies $Q(\bar{R}) \subseteq Q(\bar{R}')$.

The last component of the proof will be the following 'quantifier elimination' result for monotone queries, stating that the existence of a query satisfying certain equalities between input and output relations reduces to a boolean combination of containments between these relations.

**Proposition 33.** *Let $m > 0$ and, for every $1 \leq i \leq m$, let $\bar{\tau} \to \sigma_i$ be a type of order 1. Further let $i_1, \ldots, i_M \in \{1, \ldots, m\}$ be some indices and, for every $1 \leq j \leq M$, let $\bar{T}_j$ be a tuple of relations of type $\bar{\tau}$ and $S_j$ a relation of type $\sigma_{i_j}$. The following properties are equivalent:*

1. *there exist queries $Q_1, \ldots, Q_m$ of Positive Relational Algebra such that $Q_{i_j}(\bar{T}_j) = S_j$ for all $1 \leq j \leq M$;*
2. *for all indices $j$ and $j'$, with $1 \leq j, j' \leq M$ and $i_j = i_{j'}$, if $\bar{T}_j \subseteq \bar{T}_{j'}$, then $S_j \subseteq S_{j'}$.*

*Proof.* The implication from 1. to 2. follows trivially from the monotonicity of queries of Positive Relational Algebra. The implication from 2. to 1. is proved as follows. First, we introduce, for every index $1 \leq j \leq M$, a monotone query $P_j$ that, given a tuple $\bar{R}$ of input relations, returns either $S_j$ or the empty relation, depending on whether or not $\bar{T}_j \subseteq \bar{R}$. Note that, by construction, we have $P_j(\bar{T}_j) = S_j$. Then, we define the monotone queries $Q_1, \ldots, Q_m$ as follows. For every $1 \leq k \leq m$, $Q_k$ is the union of the queries $P_j$ over all indices $1 \leq j \leq M$ such that $i_j = k$. It is easy to check that property 2. implies $Q_{i_j}(\bar{T}_j) = S_j$ for all $1 \leq j \leq M$. □

**Remark 34.** *The above result depends heavily on the presence of relational constants. Characterizations of query definability with constant-free languages do exist – in the database community these date back to the work of Bancilhon [5] and Paredaens [41] (see also the recent work of Fletcher et al. [20], whose results bear some similarity to the proposition above). However, such characterizations are more complex, and thus query definability in these other languages cannot be reduced to a set of inclusion constraints.*

We are now ready to prove the complexity upper bound of Theorem 28 for the order-2 containment problem.

**Proposition 35.** *The order-2 containment problem with signature $\mathsf{RA}^+$ and base $\lambda\mathsf{RA}^+$ is in $\Pi_2^{\mathrm{P}}$.*

*Proof.* We fix two terms $t, t' \in \mathsf{HO}_2^{\downarrow}[\mathsf{RA}^+]$:

$$t = \lambda \mathtt{Q}_1 \ldots \lambda \mathtt{Q}_m.\ \lambda \mathtt{R}_1 \ldots \lambda \mathtt{R}_n.\ u$$
$$t' = \lambda \mathtt{Q}_1 \ldots \lambda \mathtt{Q}_m.\ \lambda \mathtt{R}_1 \ldots \lambda \mathtt{R}_n.\ u'$$

where every $\mathtt{Q}_i$ is a query variable, every $\mathtt{R}_j$ is a relational variable, and $u, u'$ are well-typed terms of order 0 using the variables $\mathtt{Q}_1, \ldots, \mathtt{Q}_m$, $\mathtt{R}_1, \ldots, \mathtt{R}_n$ and the constants of the signature $\mathsf{RA}^+$. We give a logical characterization of the non-containment relationship $t \not\leq^{\lambda\mathsf{RA}^+} t'$, that is, the existence of some queries $Q_1, \ldots, Q_m$ of Positive Relational Algebra and some relations $R_1, \ldots, R_n$ that witness $[\![t]\!](\bar{Q}, \bar{R}) \not\subseteq [\![t']\!](\bar{Q}, \bar{R})$. For this, we need to introduce new relations representing the intermediate outputs produced by the relational subterms of $u$ and $u'$ (we explain the construction for $u$ only, the one for $u'$ is similar).

We first enumerate all occurrences of query variables in $u$, say $\tilde{\mathtt{Q}}_1, \ldots, \tilde{\mathtt{Q}}_M$ (the annotation with tilde is used to distinguish variable names from variable occurrences). For each variable occurrence $\tilde{\mathtt{Q}}_j$, with $1 \leq j \leq M$, we denote by $i_j$ the index in $\{1, \ldots, m\}$ of the corresponding query variable according to the enumeration $\mathtt{Q}_1, \ldots, \mathtt{Q}_m$. In addition, we associate with each variable occurrence $\tilde{\mathtt{Q}}_j$ a relation $S_j$ of the same type as the query variable $\mathtt{Q}_{i_j}$ – this relation $S_j$ represents the result of evaluating (an instance of) $\mathtt{Q}_{i_j}$ on the arguments of that occurrence. For the sake of brevity, we let $\bar{S} = (S_1, \ldots, S_M)$.

We then consider the arguments of the occurrences of query variables in $u$. For each variable occurrence $\tilde{\mathtt{Q}}_j$, we let $\bar{u}_j = (u_{j,1}, \ldots, u_{j,\ell_j})$ be the sequence of subterms that occur in $u$ as arguments of $\tilde{\mathtt{Q}}_j$. For convenience, we also denote by $\bar{u}_0$ the sequence consisting of the single term $u$. Next, we transform each sequence of subterms $\bar{u}_j$, with $0 \leq j \leq M$, into a new sequence $\bar{v}_j$ by replacing in it every top-level occurrence of a query variable $\tilde{\mathtt{Q}}_{j'}$ with the corresponding relation $S_{j'}$. We observe that, since $t$ is in normal form, all its subterms are applied to query variables and query constants only. This means that every term of the sequence $\bar{v}_j$ can be seen as a query of Positive Relational Algebra applied to the relations $\bar{R} = (R_1, \ldots, R_n)$ and $\bar{S} = (S_1, \ldots, S_M)$. Analogous definitions are given for the objects $\bar{S}' = (S_1, \ldots, S_{M'})$, $i'_1, \ldots, i'_{M'}$, $\bar{v}'_0, \bar{v}'_1, \ldots, \bar{v}'_{M'}$, with respect to the occurrences of query variables and subterms in $u'$.

We are now ready to reduce the non-containment relationship $t \not\leq^{\lambda\mathsf{RA}^+} t'$ to the following property:

$$\exists\, Q_1, \ldots, Q_m \;\; \exists\, \bar{R}, \bar{S}, \bar{S}' \quad \bar{v}_0 \not\subseteq \bar{v}'_0 \quad \wedge \bigwedge_{1 \leq j \leq M} Q_{i_j}(\bar{v}_j) = S_j \quad \wedge \bigwedge_{1 \leq j \leq M'} Q_{i'_j}(\bar{v}'_j) = S'_j \tag{2}$$

By exploiting Proposition 33, we can get rid of the existential quantification over the queries $Q_1, \ldots, Q_m$, thus obtaining:

$$\begin{aligned} &\exists\, \bar{R}, \bar{S}, \bar{S}' \;\; \bar{v}_0 \not\subseteq \bar{v}'_0 \\ &\qquad \wedge \bigwedge_{\substack{1 \leq j \leq M \\ 1 \leq j' \leq M \\ i_j = i_{j'}}} \big(\bar{v}_j \subseteq \bar{v}_{j'} \rightarrow S_j \subseteq S_{j'}\big) \quad \wedge \bigwedge_{\substack{1 \leq j \leq M' \\ 1 \leq j' \leq M' \\ i'_j = i'_{j'}}} \big(\bar{v}'_j \subseteq \bar{v}'_{j'} \rightarrow S'_j \subseteq S'_{j'}\big) \\ &\qquad \wedge \bigwedge_{\substack{1 \leq j \leq M \\ 1 \leq j' \leq M' \\ i_j = i'_{j'}}} \big(\bar{v}_j \subseteq \bar{v}'_{j'} \rightarrow S_j \subseteq S'_{j'}\big) \quad \wedge \bigwedge_{\substack{1 \leq j \leq M' \\ 1 \leq j' \leq M \\ i'_j = i_{j'}}} \big(\bar{v}'_j \subseteq \bar{v}_{j'} \rightarrow S'_j \subseteq S_{j'}\big)\,. \end{aligned} \tag{3}$$

To ease the rewriting of the above property, it is convenient to consider the relations $\bar{R}, \bar{S}, \bar{S}'$ as a single tuple $\bar{U}$ whose elements are indexed over an appropriate set $E$ isomorphic to $\{1,\dots,n\} \uplus \{1,\dots,M\} \uplus \{1,\dots,M'\}$. Similarly, we identify any sequence of terms among $\bar{v}_0, \bar{v}_1, \dots, \bar{v}_M, \bar{v}'_0, \bar{v}'_1, \dots, \bar{v}'_{M'}$ with some term $\bar{w}_e$ defined over the relations $\bar{U}$, where $e$ is an appropriate index from $E$. In particular, we assume that $\bar{w}_{e_0}$ (resp., $\bar{w}_{e'_0}$) coincides with the singleton sequence $\bar{v}_0$ (resp., $\bar{v}'_0$). Thanks to these assumptions, the containment constraints in the last two lines of Property (3) can be indexed by pairs $(e, e')$ ranging over an appropriate subset $F$ of $E \times E$. In this way the property is shortened to

$$\exists\, \bar{U} \quad \bar{w}_{e_0} \not\subseteq \bar{w}_{e'_0} \quad \wedge \bigwedge_{(e,e') \in F} \big(\bar{w}_e \subseteq \bar{w}_{e'} \;\rightarrow\; U_e \subseteq U_{e'}\big) \tag{4}$$

We consider now maximal (consistent) sets of positive and negative containment constraints over pairs of relations $U_e, U_{e'}$, for all $(e, e') \in F$. More precisely, we consider a partition $G = (F^+, F^-)$ of $F$, with $F^+ \cup F^- = F$ and $F^- \cap F^- = \varnothing$, and we denote by $\Gamma_G$ the set that contains the positive containment constraints $U_e \subseteq U_{e'}$, for all $(e, e') \in F^+$, and the negative containment constraints $U_e \not\subseteq U_{e'}$, for all $(e, e') \in F^-$. Intuitively, the set $\Gamma_G$ describes all possible containments that can hold over a certain instance $\bar{U}$. Therefore, Property (4) holds iff there exists a partition $G = (F^+, F^-)$ of $F$ such that

$$\exists\, \bar{U} \vDash \Gamma_G \quad \bar{w}_{e_0} \not\subseteq \bar{w}_{e'_0} \quad \wedge \bigwedge_{(e,e') \in F^-} \big(\bar{w}_e \not\subseteq \bar{w}_{e'}\big) \tag{5}$$

Towards a conclusion, we recall that the terms $\bar{w}_e$ are over the signature $\mathsf{RA}^+$. In particular, a negative containment constraint of the form $\bar{w}_e \not\subseteq \bar{w}_{e'}$ holds iff there exists a conjunct $\bar{z}$ of the flattening of $\bar{w}_e$ such that $\bar{z} \not\subseteq \bar{w}_{e'}$. This shows that Property (5) above is *violated* (and hence $t \subseteq^{\lambda\mathsf{RA}^+} t'$) iff, for all partitions $G = (F^+, F^-)$ of $F$, for all pairs of indices $(e, e') \in \{(e_0, e'_0)\} \cup F^-$, and for all choices of disjuncts $\bar{z}_e$ from the flattening of $\bar{w}_e$, the following instance of the constrained disjunctive containment problem is satisfied:

$$\forall\, \bar{U} \vDash \Gamma_G \quad \bar{z}_{e_0} \subseteq \bar{w}_{e'_0} \quad \vee \bigvee_{(e,e') \in F^-} \big(\bar{z}_e \subseteq \bar{w}_{e'}\big) \tag{6}$$

The above characterization, together with Lemma 31 (which shows that a disjunct of the flattening of a term in $\mathsf{HO}_1^{\downarrow}[\mathsf{RA}^+]$ can be guessed in non-deterministic polynomial time) and Proposition 32 (which shows the NP membership of the constrained disjunctive problem with left-hand side terms in $\mathsf{HO}_1^{\downarrow}[\mathsf{SPJ}^{\mathsf{sing}}]$ and right-hand side terms in $\mathsf{HO}_1^{\downarrow}[\mathsf{RA}^+]$), proves that the problem of deciding $t \subseteq^{\lambda\mathsf{RA}^+} t'$ is in $\Pi_2^{\mathrm{P}}$. □

In the following proposition, we prove a matching $\Pi_2^{\mathrm{P}}$ lower bound for the considered order-2 containment problem. In fact, as shown in the proof, $\Pi_2^{\mathrm{P}}$-hardness holds already for containment between left-hand side order-1 terms in the signature $\mathsf{RA}^+$ and right-hand side order-1 terms in the signature $\mathsf{SPJ}^{\mathsf{sing}}$. In particular, it is worth comparing this result with Proposition 32, where the

complexity of the order-1 containment problem is shown to decrease to NP when left-hand side terms are in the signature $\mathsf{SPJ}^{\mathsf{sing}}$.

**Proposition 36.** *The order-2 containment problem with signature* $\mathsf{RA}^+$ *and base* $\lambda\mathsf{RA}^+$ *is* $\Pi_2^{\mathrm{P}}$*-hard.*

*Proof.* We actually prove the following stronger result:

**Claim.** *The containment problem for left-hand side terms in* $\mathsf{HO}_1^{\downarrow}[\mathsf{RA}^+]$ *and right-hand side terms in* $\mathsf{HO}_1^{\downarrow}[\mathsf{SPJ}^{\mathsf{sing}}]$ *is* $\Pi_2^{\mathrm{P}}$*-hard.*

Once we have proven this, the claim of the proposition follows easily by considering instances of the order-2 containment problem of the form $\lambda\mathtt{Q}.t \subseteq^{\lambda\mathsf{RA}^+} \lambda\mathtt{Q}.t'$, where $t \in \mathsf{HO}_1^{\downarrow}[\mathsf{RA}^+]$, $t' \in \mathsf{HO}_1^{\downarrow}[\mathsf{SPJ}^{\mathsf{sing}}]$, and neither $t$ nor $t'$ contain occurrences of the query variable $\mathtt{Q}$. The proof of the above claim uses the same technique as the $\Pi_2^{\mathrm{P}}$-hardness proof for the problem of deciding containment between two monotonic relational expressions, see, for instance, [46]. This claim, however, strongly relies on the use of relational constants in the left-hand side terms (on the other hand, it does not need relational constants in right-hand side terms).

We reduce from the $\forall\exists$-3CNF problem, which is known to be $\Pi_2^{\mathrm{P}}$-hard. More precisely, we fix two tuples of boolean variables $\bar{x} = (x_1, \ldots, x_m)$ and $\bar{y} = (y_1, \ldots, y_n)$ and a 3CNF formula $\bar{\alpha} = \alpha_1 \wedge \ldots \wedge \alpha_k$, where each $\alpha_i$ is a clause of the form $\alpha_i^1 \vee \alpha_i^2 \vee \alpha_i^3$ and each $\alpha_i^j$ is a literal over the variables $\bar{x}, \bar{y}$. We then reduce the problem of deciding whether for all assignments of $\bar{x}$, there exists an assignment of $\bar{y}$ satisfying $\bar{\alpha}$ to a containment problem. To do that, we see the variables $x_1, \ldots, x_m$, $y_1, \ldots, y_n$ as attributes with values ranging over $\mathbb{B} = \{0, 1\}$. Moreover, for each clause $\alpha_i$, we introduce the relational type $\tau_i = (v_i^1, v_i^2, v_i^3)$, where $v_i^j$ is the variable that appears in the literal $\alpha_i^j$. We observe that any tuple $\bar{c} \in \mathsf{dom}(\tau_i)$ represents a possible assignment for the three variables that occur in the clause $\alpha_i$ (therefore, it encodes the truth value of the clause $\alpha_i$ as well). We fix once and for all the relational constants $S_1, \ldots, S_k$ that represent the sets of all possible assignments for the variables in the clauses $\alpha_1, \ldots, \alpha_k$. Note that the natural join $S_1 \bowtie \ldots \bowtie S_k$ represents the set of all possible assignments for the variables that appear in the 3CNF formula $\bar{\alpha}$.

We can now define the left-hand side term $t$ and the right-hand side term $t'$ for the corresponding instance of the containment problem. For each clause $\alpha_i$, we list the seven (out of eight) tuples over the attributes of $\tau_i$ that induce variable assignments satisfying $\alpha_i$. Let $\bar{c}_{i,1}, \ldots, \bar{c}_{i,7}$ be these tuples. We then define the following query in the signature $\mathsf{RA}^+$ (note that the query does not use unions, but still uses relational constants):

$$t \;=\; \lambda\mathtt{R}_1 \ldots \lambda\mathtt{R}_k. \bigwedge_{\substack{1 \le i \le k \\ 1 \le j \le 7}} \pi_{\varnothing}\big(\sigma_{=\bar{c}_{i,j}}(\mathtt{R}_i)\big) \;\bowtie\; \pi_{\{x_1,\ldots,x_m\}}\big(S_1 \bowtie \ldots \bowtie S_k\big)\,.$$

Intuitively, the term $t$ above receives as input a tuple of relations $R_1, \ldots, R_k$, one for each clause of $\bar{\alpha}$, and returns either the set of all possible assignments of variables $x_1, \ldots, x_m$ or the empty set, depending on whether or not, for every $1 \le i \le k$, $R_i$ contains all tuples among $\bar{c}_{i,1}, \ldots, \bar{c}_{i,7}$ (namely, all possible ways of

satisfying the clause $\alpha_i$). As for the right-hand side term, we define the following term in the signature $\mathsf{SPJ}^{\mathsf{sing}}$ that simply projects an input assignment onto the variables $x_1, \ldots, x_m$:

$$t' \quad = \quad \lambda \mathtt{R}_1 \ldots \lambda \mathtt{R}_k. \;\; \pi_{\{x_1,\ldots,x_m\}}\big(\mathtt{R}_1 \bowtie \ldots \bowtie \mathtt{R}_k\big) \; .$$

We now prove that

$$t \subseteq t' \qquad \text{iff} \qquad \forall\, f : \mathbb{B}^{\{x_1,\ldots,x_m\}} \quad \exists\, g : \mathbb{B}^{\{y_1,\ldots,y_n\}} \quad f, g \vDash \bar{\alpha}$$

$\Rightarrow$) Assume that $t \subseteq t'$ and let $S$ be the subset of $S_1 \bowtie \ldots \bowtie S_k$ that contains all assignments satisfying $\bar{\alpha}$. For every $1 \leq i \leq k$, let $R_i$ be the projection of $S$ over the attributes of $\tau_i$. Clearly, each relation $R_i$ contains all the possible assignments that satisfy $\alpha_i$. In particular, the term $\pi_{\varnothing}\big(\sigma_{=\bar{c}_{i,j}}(R_i)\big)$ evaluates to $\mathsf{true}$, for every $1 \leq i \leq k$ and every $1 \leq j \leq 7$. Hence, $t(R_1, \ldots, R_k)$ evaluates to the set $T$ of all possible assignments for the variables $x_1, \ldots, x_m$ that appear in the clauses $\alpha_1, \ldots, \alpha_k$. Since $t \subseteq t'$, we know that $t'(R_1, \ldots, R_k)$ contains all the tuples from $T$. This shows that every assignment $f$ for $x_1, \ldots, x_m$ can be extended by an assignment $g$ for $y_1, \ldots, y_n$ so as to satisfy $\bar{\alpha}$.

$\Leftarrow$) Suppose that every assignment $f$ for $x_1, \ldots, x_m$ can be extended by an assignment $g$ for $y_1, \ldots, y_n$ in such a way that $\bar{\alpha}$ is satisfied. Fix some generic relations $R_1, \ldots, R_k$ of type $\tau_1, \ldots, \tau_k$ and consider the result of evaluating $t$ on these relations. If $t(R_1, \ldots, R_k)$ returns the empty set, then $[\![t]\!](R_1, \ldots, R_k) \subseteq [\![t']\!](R_1, \ldots, R_k)$ holds trivially. Otherwise, let $\bar{c}$ be any tuple in the set $T = [\![t]\!](R_1, \ldots, R_k)$. By construction, the tuple $\bar{c}$ represents some assignment $f$ for the variables $x_1, \ldots, x_m$. From the initial assumptions, we know that $f$ can be extended by an assignment $g$ on $y_1, \ldots, y_n$ in such a way that $\bar{\alpha}$ is satisfied. Moreover, we know from the fact that $T = [\![t]\!](R_1, \ldots, R_k)$ is non-empty that the tuple that encodes the complete assignment $f, g$ belongs to the set $R_1 \bowtie \ldots \bowtie R_k$. In particular, $\bar{c}$ can be viewed as the projection of $f, g$ over the variables $x_1, \ldots, x_m$, and hence $\bar{c} \in [\![t']\!](R_1, \ldots, R_k)$.

This completes the proof of the proposition. □

Proposition 35 and Proposition 36 together give precisely the claim of Theorem 28.

Below, we show how to modify the proof of Theorem 28 to get similar complexity bounds for containment of terms of order 2 over the signature $\mathsf{RA}^{+}$, but under a slightly different base. Formally, let $\mathsf{RA}^{+,\neq}$ denote the signature that extends $\mathsf{RA}^{+}$ with selection operators that use equalities and inequalities over attributes and constants (e.g., $\sigma_{a \neq 1}$), and let $\lambda\mathsf{RA}^{+,\neq}$ be the corresponding base that restricts all formal arguments of order 1 to range over queries definable in the signature $\mathsf{RA}^{+,\neq}$.

**Theorem 37.** *The order-2 containment problem with signature* $\mathsf{RA}^{+}$ *and base* $\lambda\mathsf{RA}^{+,\neq}$*, namely, the problem of deciding whether* $t \subseteq^{\lambda\mathsf{RA}^{+,\neq}} t'$ *for terms* $t, t' \in \mathsf{HO}_2^{\downarrow}[\mathsf{RA}^{+}]$*, is* $\Pi_2^{\mathrm{P}}$*-complete.*

*Proof.* Recall that the $\Pi_2^{\mathrm{P}}$-hardness result is derived from Proposition 36 by considering the order-1 containment problem. This hardness result is thus independent of the choice of the base, and thus it holds also for the containment problem between terms in $\mathsf{HO}_2^{\downarrow}[\mathsf{RA}^+]$ with base $\lambda\mathsf{RA}^{+,\neq}$.

For the upper bound, we recall that the proof of Proposition 35 is based on a few crucial properties, that is: (i) the existence of queries of the Positive Relational Algebra satisfying certain equalities between input and output relations reduces to a boolean combination of containments between these relations (cf. Proposition 33), (ii) the constrained disjunctive containment problem for left-hand side $\mathsf{SPJ}^{\mathsf{sing}}$-terms and right-hand side $\mathsf{RA}^+$-terms is in NP. Further recall that the first property follows from the fact that the considered queries are monotone. Thus, the same property holds when we consider queries definable over the larger signature $\mathsf{RA}^{+,\neq}$. This shows that the $\Pi_2^{\mathrm{P}}$ upper bound holds also for the order-2 containment problem with signature $\mathsf{RA}^+$ and base $\lambda\mathsf{RA}^{+,\neq}$. □

Moreover, recall that the normal form of a term $t$ of degree $d$ can be computed in time $\mathsf{tow}_{d+1}\big(\mathcal{O}(|t|)\big)$. Pairing this with Theorem 28 (resp., Theorem 37) proves that the containment problem for *unnormalized* terms of bounded degree under the base $\lambda\mathsf{RA}^+$ (resp., $\lambda\mathsf{RA}^{+,\neq}$) has elementary complexity:

**Corollary 38.** *For every $d \geq 1$, the containment problem for terms in $\mathsf{HO}_2^d[\mathsf{RA}^+]$ with base $\lambda\mathsf{RA}^+$ or $\lambda\mathsf{RA}^{+,\neq}$ is in $\Pi_2^{(d+1)\text{-Exp}}$.*

**Adding dependencies.** We now consider order-2 containment relative to *integrity constraints*. We focus on two widely-studied constraint classes, namely, functional dependencies and inclusion dependencies [1]. The containment problem for conjunctive queries under sets of functional dependencies has been deeply investigated starting from [2] and it is known to be NP-complete.

Below, given two terms $t, t' \in \mathsf{HO}_2^{\downarrow}[\mathsf{RA}^+]$ of the same type and given a set $\Delta$ of constraints (e.g., functional dependencies) over the relational arguments of $t$ and $t'$, we write $t \subseteq_{\Delta}^{\lambda\mathsf{RA}^+} t'$ iff for all inputs $\bar{Q}, \bar{R}$ that satisfy the constraints in $\Delta$, we have $[\![t]\!](\bar{Q}, \bar{R}) \subseteq [\![t']\!](\bar{Q}, \bar{R})$. We can easily extend Theorem 28 to this new setting:

**Theorem 39.** *The problem of deciding whether $t \subseteq_{\Delta}^{\lambda\mathsf{RA}^+} t'$ for terms $t, t' \in \mathsf{HO}_2^{\downarrow}[\mathsf{RA}^+]$ and a set $\Delta$ of functional dependencies is $\Pi_2^{\mathrm{P}}$-complete.*

The proof of the complexity upper bound goes along the same lines of the proof of Proposition 35. More precisely, we first exploit Proposition 33 (which is independent of the presence of constraints on the relations) to reduce the containment problem for higher-order queries to the problem of universally guessing and deciding suitable instances of the disjunctive containment problem involving left-hand side terms in $\mathsf{HO}_1^{\downarrow}[\mathsf{SPJ}^{\mathsf{sing}}]$ and right-hand side terms in $\mathsf{HO}_1^{\downarrow}[\mathsf{RA}^+]$, under positive and negative containment constraints and the additional functional dependencies. We then reduce the latter variant of the constrained disjunctive containment problem a simpler disjunction-free containment problem and finally argue that the problem is in NP:

**Lemma 40.** *The constrained disjunctive containment problem for left-hand side terms in* $\mathsf{HO}_1^{\downarrow}[\mathsf{SPJ}^{\mathsf{sing}}]$ *and right-hand side terms in* $\mathsf{HO}_1^{\downarrow}[\mathsf{RA}^{+}]$*, under positive and negative containment constraints and functional dependencies, reduces in polynomial time to the (disjunction-free) constrained containment problem for left-hand side terms in* $\mathsf{HO}_1^{\downarrow}[\mathsf{SPJ}^{\mathsf{sing}}]$ *and right-hand side terms in* $\mathsf{HO}_1^{\downarrow}[\mathsf{RA}^{+}]$*, under positive containment constraints and functional dependencies.*

The proof of the above result is almost the same as that of Lemma 30 and thus omitted. The last ingredient for claiming Theorem 39 is given in the next proposition, whose proof is very similar to that of Proposition 32:

**Proposition 41.** *The constrained disjunctive containment problem for left-hand side terms in* $\mathsf{HO}_1^{\downarrow}[\mathsf{SPJ}^{\mathsf{sing}}]$ *and right-hand side terms in* $\mathsf{HO}_1^{\downarrow}[\mathsf{RA}^{+}]$ *under positive and negative containment constraints and functional dependencies, is* NP*-complete.*

*Proof.* NP-hardness holds already for the (unconstrained disjunction-free) containment problem between conjunctive queries. For the complexity upper bound, thanks to Lemma 40, it is sufficient to prove NP membership of the disjunction-free containment problem under positive constraints and functional dependencies. As usual, we restrict to terms defining boolean queries.

Let us consider a generic instance of the constrained containment problem, which consists of a term $t \in \mathsf{HO}_1^{\downarrow}[\mathsf{SPJ}^{\mathsf{sing}}]$ of query type $(\tau_1 \times \ldots \times \tau_m) \to ()$, a term $t' \in \mathsf{HO}_1^{\downarrow}[\mathsf{RA}^{+}]$ of the same type, a set $\Gamma$ of positive containment constraints, and a set $\Delta$ of functional dependencies over relations $R_1, \ldots, R_m$ of types $\tau_1, \ldots, \tau_m$, respectively. Let $Q = [\![t]\!]$ be the conjunctive query defined by $t$. We transform $Q$ into a new conjunctive query $Q^*$ by applying a variant of the chase procedure for inclusion and functional dependencies [1, 30]. Formally, $Q^*$ is obtained by applying exhaustively first Rule 1. and then Rule 2. below:

1. If $R_i \subseteq R_j$ is a containment constraint in $\Gamma$, $\bar{c}$ is a tuple associating variables or constants with the attributes of $\tau_i$, and $R_i(\bar{c})$ appears as a predicate in the current expansion of $Q$, then add the new conjunct $R_j(\bar{c})$ to $Q$.

2. If $R_i.A \to R_i.B$ is a functional dependency of $\Delta$, $\bar{c}$ and $\bar{c}'$ are tuples associating variables or constants with the attributes of $\tau_i$, and the current expansion of $Q$ contains two conjuncts of the form $R_i(\bar{c})$ and $R_i(\bar{c}')$, with $\pi_A(\bar{c}) = \pi_A(\bar{c}')$, then identify, by using either equalities or substitutions, any two variables/constants $\bar{c}(b)$ and $\bar{c}'(b)$ associated with the same attribute $b \in B$.

As usual, both the expansion $Q^*$ and its canonical model $\bar{R}^{Q^*}$ can be computed from $t$ in polynomial time. Moreover, by using arguments similar to that of the proof of Proposition 32, we can reduce the constrained containment problem to a query evaluation problem:

$$t \subseteq_{\Gamma,\Delta} t' \qquad \text{iff} \qquad [\![t']\!]\left(\bar{R}^{Q^*}\right) = \mathsf{true}.$$

Finally, in order to decide the latter property it is sufficient to guess (i) a disjunct $\tilde{t}'_i$ of the flattening of $t'$ (see Lemma 31) and (ii) a homomorphism from $\bar{R}^{Q'_i}$ to $\bar{R}^{Q^*}$, where $Q'_i = [\![\tilde{t}'_i]\!]$, so as to witness $[\![t']\!](\bar{R}^{Q^*}) = \mathsf{true}$, and hence $t \subseteq_{\Gamma,\Delta} t'$. □

We now consider the order-2 containment in the presence of inclusion dependencies.

**Theorem 42.** *The problem of deciding whether* $t \subseteq_{\Delta}^{\lambda\mathsf{RA}^+} t'$ *for terms* $t, t' \in \mathsf{HO}_2^{\downarrow}[\mathsf{RA}^+]$ *and a set* $\Delta$ *of inclusion dependencies is* PSPACE*-complete.*

*Proof.* It is known that the containment problem between two conjunctive queries under a set $\Delta$ of inclusion dependencies is PSPACE-hard (see, for instance, [11]). This lower bound can be immediately transferred to the order-2 containment problem under inclusion dependencies.

Below, we prove the PSPACE upper bound. Using the same transformation as in the proof of Theorem 28 (cf. Equation (6)), we reduce the order-2 containment $t \subseteq_{\Delta}^{\lambda\mathsf{RA}^+} t'$ under a set $\Delta$ of inclusion dependencies to the problem of universally guessing and deciding instances of a disjunctive containment problem of the form:

$$\forall\, \bar{U} \vDash \Gamma^+, \Gamma^-, \Delta \quad \bigvee_i \Big( [\![z_i]\!](\bar{U}) \subseteq [\![w_i]\!](\bar{U}) \Big)$$

where $\Gamma^+$ and $\Gamma^-$ are sets of positive and negative containment constraints, the $z_i$'s are terms in $\mathsf{HO}_1^{\downarrow}[\mathsf{SPJ}^{\mathsf{sing}}]$, and the $w_i$'s are terms in $\mathsf{HO}_1^{\downarrow}[\mathsf{RA}^+]$.

We observe that positive containment constraints in $\Gamma^+$ are special forms of inclusion dependencies, and hence can be absorbed in $\Delta$. Moreover, by a straightforward generalization of the proof of Proposition 32, the constrained disjunctive containment problem for left-hand side terms in $\mathsf{HO}_1^{\downarrow}[\mathsf{SPJ}^{\mathsf{sing}}]$ and right-hand side terms in $\mathsf{HO}_1^{\downarrow}[\mathsf{RA}^+]$ under negative containment constraints and inclusion dependencies reduces to the (disjunction-free) constrained containment problem for left-hand side terms in $\mathsf{HO}_1^{\downarrow}[\mathsf{SPJ}^{\mathsf{sing}}]$ and right-hand side terms in $\mathsf{HO}_1^{\downarrow}[\mathsf{RA}^+]$ under inclusion dependencies only. Finally, the latter problem can be solved in polynomial space by guessing a disjunct of the flattening of the right-hand side term and by deciding a classical containment problem between conjunctive queries under inclusion dependencies, which is known to be in PSPACE [30]. □

## 7. Conclusions and future work

We have studied a query language that is obtained from combining Relational Algebra and simply-typed $\lambda$-calculus, and that can be used to define both ordinary queries and query functionals. This formalism has two main advantages: the output of a query transformation depends only on the semantics of the formal arguments (e.g., input queries), and, even for ordinary queries, the formalism is often more succinct than others.

In the first part, we have considered the problem of evaluating terms of order 0 in our language. Since terms that use $\lambda$-abstractions on query variables are in general more succinct than simple terms of Relational Algebra, one should expect that the complexity of the evaluation problem increases with the degree of the terms. We provided tight complexity bounds for the evaluation problem for terms of different degrees that show that to each increase of the degree corresponds a jump in the complexity hierarchy. Even though this implies a non-elementary lower bound when the degree is not fixed, we have identified sub-cases where the evaluation problem becomes more tractable, e.g., not harder than the analogous problem for Non-recursive Datalog programs.

In the second part, we have analysed the complexity of the containment problem for terms in normal form of order 1, as well as a natural generalization of this problem for terms in normal form of order 2. In the former case, the terms define ordinary queries over (tuples of) relations; whereas in the latter case, terms define meta-queries with inputs consisting of both relations and queries. Of course, the complexity (and even the decidability) of the containment problem for order-2 terms depends heavily on underlying signature of the terms and base for the query variables. However, when we choose the Positive Relational Algebra as the underlying signature and base for the query variables, the containment problem is proven to be $\Pi_2^{\mathrm{P}}$-complete, thus not more difficult than the classical containment problem for terms of order 1. We leave open the study of the decidability and complexity of the containment problem for order-2 terms over the signature and base SPJ. The exact complexity of the containment problem for terms that are *not* in normal form remains also open.

The higher-order query language that we defined does not allow terms receiving inputs of different types. There are a number of works studying polymorphic type inference for relational data, e.g., [38, 10, 54]. For example, a higher-order function may return the projection onto attribute $A$ of an input relation, and in that case one could let the type of the input be unspecified, provided that it contains at least the attribute $A$. A higher-order language that supports this construction would help users to write queries in a more flexible way.

Van den Bussche and his colleagues consider the "well-definedness problem" for languages without higher-order variables, which is a more lenient and less syntactic condition than typeability, obtained by allowing some subterms to be untypeable [55, 51, 53, 52]. The analogous problem for higher-order terms remains to be considered. **Acknowledgments:** We would like to thank Jan Van den Bussche for helpful discussions on the conference version, and the anonymous reviewers of PODS, ICDT, and Information and Computation for many useful remarks. Benedikt's work was supported by the Engineering and Physical Science Research Council project "Enforcement of Constraints on XML Streams" EP/G004021/1.

## References

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

[2] A. V. Aho, Y. Sagiv, and J. D. Ullman. Efficient optimization of a class of relational expressions. *ACM Transactions on Database Systems*, 4(4):435–454, 1979.

[3] S. Amer-Yahia, S. Cho, and D. Srivastava. Tree pattern relaxation. In *EDBT*, 2002.

[4] J. Avigad. Eliminating Definitions and Skolem Functions in First-order Logic. *ACM Transactions on Computational Logic*, 4(3):402–415, 2003.

[5] F. Bancilhon. On the completeness of query languages for relational data bases. In *MFCS*, 1978.

[6] M. Benedikt and G. Gottlob. The impact of virtual views on containment. In *VLDB*, pages 297–308, 2010.

[7] M. Benedikt and C. Koch. From XQuery to relational logics. *ACM Transactions on Database Systems*, 34(4):1–48, 2009.

[8] M. Benedikt, G. Puppis, and H. Vu. Positive higher-order queries. In *PODS*, 2010.

[9] H. Björklund, W. Martens, and T. Schwentick. Conjunctive query containment over trees. *Journal of Computer and System Sciences*, 77(3):450 – 472, 2011.

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

[11] M. Casanova, R. Fagin, and C. Papadimitriou. Inclusion dependencies and their interaction with functional dependencies. *Journal of Computer and System Sciences*, 28(1):29–59, 1984.

[12] B. Cautis, A. Deutsch, and N. Onose. Querying Data Sources that Export Infinite sets of Views. In *ICDT*, 2009.

[13] A. K. Chandra and P. M. Merlin. Optimal implementation of conjunctive queries in relational databases. In *STOC*, 1977.

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

[15] E. Cooper. The script-writers dream: How to write great SQL in your own language, and be sure it will succeed. In *DBPL*, 2009.

[16] E. Cooper, S. Lindley, P. Wadler, and J. Yallop. Links: Web programming without tiers. In *FMCO*, 2007.

[17] E. Dantsin, T. Eiter, G. Gottlob, and A. Voronkov. Complexity and expressive power of logic programming. *ACM Computing Surveys*, 33(3):374–425, 2001.

[18] X. Dong, A. Y. Halevy, and I. Tatarinov. Containment of nested XML queries. In *VLDB*, 2004.

[19] H.-D. Ebbinghaus and J. Flum. *Finite Model Theory*. Springer, 1999.

[20] G. H. L. Fletcher, M. Gyssens, J. Paredaens, and D. V. Gucht. On the expressive power of the Relational Algebra on finite sets of relation pairs. *IEEE Transactions on Knowledge and Data Engineering*, 21(6):939–942, 2009.

[21] J.Y. Girard, P. Taylor, and Y. Lafont. *Proofs and types*. Cambridge University Press, 1989.

[22] G. Gottlob and C. Papadimitriou. On the complexity of single-rule datalog queries. *Information and Computation*, 183(1):104–122, 2003.

[23] T. Grust, M. Mayr, J. Rittinger, and T. Schreiber. FERRY: Database-supported program execution. In *SIGMOD*, 2009.

[24] G. Hillebrand. *Finite Model Theory in the Simply Typed Lambda Calculus*. PhD thesis, Brown University, 1994.

[25] G. Hillebrand and P. Kanellakis. Functional database query languages as typed lambda calculi of fixed order. In *PODS*, 1994.

[26] G. Hillebrand and P. Kanellakis. On the expressive power of simply typed and let-polymorphic lambda calculi. In *LICS*, 1996.

[27] G. Hillebrand, P. Kanellakis, and H. Mairson. Database query languages embedded in the typed lambda calculus. In *LICS*, 1993.

[28] J. R. Hindley. *Basic Simple Type Theory*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2008.

[29] N. Immerman. Relational queries computable in polynomial time. *Information and control*, 68(1-3):86–104, 1986.

[30] D. S. Johnson and A. C. Klug. Testing containment of conjunctive queries under functional and inclusion dependencies. *Journal of Computer and System Sciences*, 28(1):167–189, 1984.

[31] C. Koch. On the complexity of nonrecursive XQuery and functional query languages on complex values. *ACM Transactions on Database Systems*, 31(4):1215–1256, 2006.

[32] N. Koudas, C. Li, A. Tung, and R. Vernica. Relaxing join and selection queries. In *VLDB*, 2006.

[33] D. Kozen. Lower bounds for natural proof systems. In *FOCS*, pages 254–266, 1977.

[34] L. V. S. Lakshmanan, F. Sadri, and I. N. Subramanian. On the logical foundations of schema integration and evolution in heterogeneous database systems. In *DOOD*, 1993.

[35] A. Levy, A. Rajaraman, and J. Ullman. Answering Queries using Limited External Query Processors. In *PODS*, 1996.

[36] A. Y. Levy and D. Suciu. Deciding containment for queries with complex objects (extended abstract). In *PODS*, 1997.

[37] H. G. Mairson. A simple proof of a theorem of Statman. *Theoretical Computer Science*, 103(2):387–394, 1992.

[38] R. Milner. A theory of type polymorphism in programming. *Journal of computer and system sciences*, 17(3):348–375, 1978.

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

[40] A. Ohori, P. Buneman, and V. Breazu-Tannen. Database programming in Machiavelli–a polymorphic language with static type inference. In *SIGMOD*, 1989.

[41] J. Paredaens. On the expressive power of the relational algebra. *Information Processing Letters*, 7(2):107–111, 1978.

[42] J. Paredaens and D. Van Gucht. Converting nested algebra expressions into flat algebra expressions. *ACM Transactions on Database Systems*, 17(1):65–93, 1992.

[43] J. Robie, D. Chamberlin, M. Dyck, and J. Snelson. XQuery 3.0: An XML Query Language. http://www.w3.org/TR/2011/WD-xquery-30-20111213, 2010.

[44] K. A. Ross. Relations with relation names as arguments: Algebra and calculus. In *PODS*, 1992.

[45] K.A. Ross. On negation in HiLog. *The Journal of Logic Programming*, 18(1):27–53, 1994.

[46] Y. Sagiv and M. Yannakakis. Equivalences among relational expressions with the union and difference operators. *Journal of the ACM*, 27(4):633–655, 1980.

[47] A. Schubert. The complexity of $\beta$-reduction in low orders. In *TLCA*, 2001.

[48] R. Statman. The typed $\lambda$-calculus is not elementary recursive. *Theoretical Computer Science*, 9(1):73–81, 1979.

[49] V. Tannen, P. Buneman, and L. Wong. Naturally embedded query languages. In *ICDT*, 1992.

[50] A. Ulrich. A FERRY-based query backend for the LINKS programming language. Master's thesis, University of Tübingen, 2011.

[51] J. Van den Bussche, D. Van Gucht, and S. Vansummeren. Well-definedness and semantic type-checking in the nested relational calculus and XQuery. In *ICDT*, pages 99–113, 2005.

[52] J. Van den Bussche, D. Van Gucht, and S. Vansummeren. A crash course on database queries. In *PODS*, 2007.

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

[54] J. Van den Bussche and E. Waller. Polymorphic type inference for the Relational Algebra. *Journal of Computer and System Sciences*, 64(3):694–718, 2002.

[55] S. Vansummeren. Deciding well-definedness of XQuery fragments. In *PODS*, pages 37–48, 2005.

[56] M. Y. Vardi. The complexity of relational query languages. In *STOC*, pages 137–146, 1982.

[57] V. Vassalos and Y. Papakonstantinou. Expressive capabilities description languages and query rewriting algorithms. *The Journal of Logic Programming*, 43(1):75–122, 2000.

[58] S. Vorobyov and A. Voronkov. Complexity of nonrecursive logic programs with complex values. In *PODS*, 1998.

[59] H. Vu and M. Benedikt. Complexity of Higher-Order Queries. In *ICDT*, 2011.

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