

# Complexity of Higher-Order Queries

Huy Vu  
Oxford University  
Computing Laboratory  
Parks Road, Oxford, UK  
huy.vu@comlab.ox.ac.uk

Michael Benedikt  
Oxford University  
Computing Laboratory  
Parks Road, Oxford, UK  
michael.benedikt@comlab.ox.ac.uk

## ABSTRACT

While relational algebra and calculus are a well-established foundation for classical database query languages, it is less clear what the analog is for higher-order functions, such as query transformations. Here we study a natural way to add higher-order functionality to query languages, by adding database query operators to the  $\lambda$ -calculus as constants. This framework, which we refer to as  *$\lambda$ -embedded query languages*, was introduced in [BPV10]. That work had a restricted focus: the containment and equivalence problems for query-to-query functions, in the case where only positive relational operators are allowed as constants. In this work we take an in-depth look at the most basic issue for such languages: the evaluation problem. We give a full picture of the complexity of evaluation for  $\lambda$ -embedded query languages, looking at a number of variations: with negation and without; with only relational algebra operators, and also with a recursion mechanism in the form of a query iteration operator; in a strongly-typed calculus as well as a weakly-typed one. We give tight bounds on both the combined complexity and the query complexity of evaluation in all these settings, in the process explaining connections with Datalog and prior work on  $\lambda$ -calculus evaluation.

## Categories and Subject Descriptors

H.2.3 [Database Management]: Logical Design, Languages—data models, query languages; F.2.0 [Analysis of Algorithms and Problem Complexity]: General

## General Terms

Theory

## 1. INTRODUCTION

Higher-order functions play a fundamental role in computer science; they are critical to functional programs, and in object-oriented programming they play a key role in encapsulation. In database systems they have appeared in iso-

lation at several points: query-transformation plays a role in numerous aspects of databases, including data integration [LRU96], access control [FGK07], and privacy [LHR<sup>+</sup>10]. In the context of nested-relational and functional query languages, the ability to create and pass functions using data plays a role; the role of higher-order functions is becoming more prominent within XML query languages, with the next version of the XQuery standard offering explicit support for higher-order features [RCDS09].

Still, the combination of database queries and higher-order functions has not been studied in its own right: in prior work within the database community it appears in conjunction with other language features and in restricted settings. Within finite model theory the ability to express database queries within higher-order functions has been studied (see Section 7 for a discussion), but not their combination. In [BPV10], a straightforward framework for combining relational algebra with higher-order functional languages is defined, which we refer to as  *$\lambda$ -embedded query languages*: it is exactly the simply-typed  $\lambda$ -calculus with database operators as “constants” (that is, as built-in functions).

**EXAMPLE 1.** *Consider the situation where an information source has a relation instance  $D_0$ . The source exposes an interface to query  $D_0$ . But for security reasons, when the source receives a query  $Q$ , it returns the result of  $Q$  on a selection of  $D_0$  and returns only two of the columns,  $a$  and  $b$ . This could be implemented via the following higher-order query*

$$\tau_0 = \lambda Q. \pi_{a,b} Q(\sigma_c D_0)$$

Our prior work [BPV10] focuses on a restricted case of this framework: where the terms are “degree 2” (variables range over either databases or queries), and the constants come from positive relational algebra. The topic of the prior paper is equivalence and containment of terms. We first showed that the equivalence problem between terms that evaluate to (ordinary relational) queries is decidable, isolating the complexity for several subclasses. The main result of [BPV10] is decidability of a notion of equivalence over terms which evaluate to query functionals.

In this paper, we give a full picture of the most basic problem concerning terms in higher-order query languages: evaluation of “order 0 terms”: terms that evaluate to a database instance. We study this not only for positive 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 1”: those that have variables ranging only over databases. For terms with only pos-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ICDT 2011, March 21–23, 2011, Uppsala, Sweden.

Copyright 2011 ACM 978-1-4503-0529-7/11/0003 ...\$10.00.

itive operators and only database variables, it was shown in [BPV10] that this language is essentially the same as non-recursive Datalog. Here we extend this to the more general setting of all relational operators, and to fixpoint operators, showing equivalence to variants of Datalog. This allows us to read off the complexity of evaluation via reduction to known results about Datalog.

We then extend to the first higher-order case: “degree 2”, where terms can have variables ranging over queries. Here we get tight bounds on the complexity of evaluation through combining an analysis of the complexity of classical  $\beta$ -reduction with the results on degree 1. Building on the degree 2 case, we determine the complexity for general terms, which can abstract over objects of any order. We use a technique inspired by Hillebrand and Kannellakis [HK96]. Our results show that the complexity is non-elementary for the general calculus.

Having found the worst-case complexity for general terms, we turn to cases that have lower complexity. For example, we show that the complexity reduces drastically when we restrict the nesting of higher-order variables in terms.

All of these results are for a “strongly-typed” version of the  $\lambda$ -calculus, in which all operators and variables must be annotated with correct types. We look at a weaker “implicitly-typed” version, and show that the complexity does not change, even though we can now build queries in which the arity of the output can grow exponentially with the input database.

**Organization:** Section 2 defines the higher-order languages we investigate, along with relevant database background. Sections 3 gives the complexity of evaluation for terms that have  $\lambda$ -abstractions only over relations and queries. Section 4 deals with evaluation for arbitrary degree for the strongly-typed language, while Section 5 isolates sublanguages with lower complexity. Section 6 studies the weakly-typed variant of the language. Section 7 overviews related work, while Section 8 gives conclusions.

## 2. DEFINITIONS

We begin by defining the higher order language studied in this work.

### Types.

We fix an infinite linearly-ordered set of *attribute names* (or *attributes*). We associate with each attribute name  $a_i$  a range  $Dom(a_i)$  of possible values, called the *attribute range* of  $a_i$ . For simplicity, we assume all attribute ranges are the integers  $\mathbb{Z}$ .

Next we will define the *types* along with their *order*, and their *domain*.

The basic types are the *relational types* each given by a (possibly empty) set of attribute names,  $\mathcal{T} = (a_1, \dots, a_m)$ . A tuple of a relational type is (as usual) a function from the attribute names to the attribute range, while an instance of a relational type is a set of tuples. The type corresponding to no attributes is denoted  $\{\}$ . We manipulate relational types by using the standard operations on tuples, such as concatenation  $\mathcal{T} + \mathcal{T}'$  (assuming no overlap of  $\mathcal{T}$  and  $\mathcal{T}'$ ) and the projection  $\pi_A(\mathcal{T})$ , for a given set  $A$  of attributes in  $\mathcal{T}$ . The domain of a relational type is the collection of all finite instances over the type.

The order of any relational type is 0.

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 denote with the boolean value **false**, and the singleton  $\{\}$ , which we denote with **true**.

We will sometimes use positional notation for attributes, particularly in queries; in a tuple  $t$  whose type is  $a_1 \dots a_m$ , ordered using the fixed ordering on attribute names, we write  $t.i$  to denote  $t.a_i$ .

### 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  $\mathcal{T}, \mathcal{T}'$  are types over the domain  $\mathcal{D}, \mathcal{D}'$ , then  $\mathcal{T} \rightarrow \mathcal{T}'$  is a type whose domain is  $(\mathcal{D}')^{\mathcal{D}}$ , the set of functions from  $\mathcal{D}$  to  $\mathcal{D}'$ , and whose order is  $\text{order}(\mathcal{T} \rightarrow \mathcal{T}') = \max(\text{order}(\mathcal{T}) + 1, \text{order}(\mathcal{T}'))$ .

We abbreviate a type of the form  $\mathcal{T}_1 \rightarrow \dots \rightarrow \mathcal{T}_m \rightarrow \mathcal{T}'$  as  $(\mathcal{T}_1 \times \dots \times \mathcal{T}_m) \rightarrow \mathcal{T}'$  (an abbreviation only, since we have no product operation on types). Similarly we will write elements of such types in their curried form.

Order 1 types are often called *query types*. We will be interested in the evaluation problem for terms of query type, where  $\mathcal{T}'$  above is the boolean type; we call the types of such terms *boolean query types*.

### Constants.

We will fix a set of constants of each type  $\mathcal{T}$ . Constants can be thought of as specific instance of the given type; formally, the semantics is defined with respect to an interpretation of each constant symbol by an object of the appropriate type; but we will often abuse notation by identifying the constant and the object. We study the following sets of constants:

- All of our signatures will include constants for all instances, referred to as *relational constants*.
- Our signatures will differ on the set of order 1 constants – i.e. *query constants*.
  - We use  $\text{RA}^+$  to denote the operators of Positive Relational Algebra, which contains the following constants: (i) for each relational type  $\mathcal{T}$  containing attribute  $a$  the unary rename operator  $\rho_{a/b}$ , which renames the attribute  $a$  by  $b$  in a given input relation; (ii) for each relational type containing attributes  $A$  the unary operator  $\pi_A$ , which projects an input relation into the subset  $A$  of its attributes; (iii) for each relational type  $\mathcal{T}$  the unary operator  $\sigma_c$ , which selects a subset of the tuples from a given relation according to the condition  $c$  specifying equalities between attributes and constants or attributes and attributes; (iv) for any two types  $\mathcal{T}$  and  $\mathcal{T}'$  the binary operator  $\boxtimes$ , which returns the cartesian product of two input relations followed by a selection of the tuples that have the same values on the same attribute names; (v) for any type  $\mathcal{T}$  the binary operator  $\cup$ , which returns the union of two input relations of type  $\mathcal{T}$ .

Although we will state complexity results for  $\text{RA}^+$ , union is never essential for the hardness results.

- RA extends  $\text{RA}^+$  with the usual difference operator  $\setminus$ , again parameterized by a given relational type.
- We also consider a signature  $\text{RA}_x$  whose query constants include the operators  $\pi_B$  for every finite set of attributes  $B$ ,  $\sigma_c$  for each condition,  $X$ , and  $\setminus$ . Note that here we do not have a distinct operator for each type. These operators will work on attributes denoted “positionally”. Formally these operations will only be well-defined on inputs whose signatures are  $(a_1 \dots a_n)$  where the  $a_i$  are an initial segment of the attribute names (briefly “initial-segment instances”). Projection on a subset consisting of  $k$  attributes will automatically rename the resulting tuples to be of type  $(a_1 \dots a_k)$ . We use  $X$  for the usual cartesian product operation on positional relations – we reserve  $\times$  for the abbreviation used for types.  $X$  is the curried form of the binary function taking an instance for schema  $(a_1 \dots a_n)$  and an instance for  $(a_1 \dots a_{n+k})$ , returning the product instance of type  $(a_1 \dots a_{n+k})$ . We can identify attribute names in projection and selection with integers, writing them in binary.

We supplement this signature with a variation on projection where we also allow the variation when  $B$  is a range of the form  $[p_1, p_2]$  which projects all attributes whose positions are in between  $p_1$  and  $p_2$ .

The signature  $\text{RA}_x$  will be studied for the weakly-typed calculus, where the number of attributes built up by terms can be large.

- Lastly, we will consider the impact of order 2 constants (that is, query to query functionals), by adding to the relational algebra signature RA the inflationary fixed point operators,  $\text{ifp}$ . For  $\mathcal{T}_1 \dots \mathcal{T}_m, \mathcal{T}'_1 \dots \mathcal{T}'_n$  relational types, let  $U_i$  for  $i \leq m$  abbreviate the query type  $(\mathcal{T}_1 \times \dots \times \mathcal{T}_m \times \mathcal{T}'_1 \times \dots \times \mathcal{T}'_n) \rightarrow \mathcal{T}_i$ .

Then we have an operator  $\text{ifp}$  having type  $U_1 \times \dots \times U_m \times \mathcal{T}'_1 \times \dots \times \mathcal{T}'_n \rightarrow \mathcal{T}_1$ . The output of  $\text{ifp}$  given queries  $Q_1 \dots Q_m$  with  $Q_i$  of type  $U_i$  and relation instances  $I_1 \dots I_n$  is determined by taking the limit of the sequences  $R_j^i$ , where the doubly-indexed sequence of instances  $R_j^i$  for  $i \leq m$  and  $j$  a number is formed as follows:  $R_0^i = \emptyset$  for all  $i \leq m$  and  $R_{j+1}^i = Q_i(R_j^1, \dots, R_j^m, I_1 \dots I_n) \cup R_j^i$ .

### Simply typed terms.

Higher-order *terms* are built up from constants in  $\mathcal{F}$  and variables in  $\mathcal{X}$  by using the operations of abstraction and application:

- every constant is a term of the constant’s type;
- if  $X$  is a variable of type  $\mathcal{T}$  and  $\rho$  is a term of type  $\mathcal{T}'$ , then  $\lambda X. \rho$  is a term of type  $\mathcal{T} \rightarrow \mathcal{T}'$ ;
- $\tau$  is a term of type  $\mathcal{T} \rightarrow \mathcal{T}'$  and  $\rho$  is a term of type  $\mathcal{T}$ , then  $\tau(\rho)$  is a term of type  $\mathcal{T}'$ .

We say that a term  $\tau$  is *closed* if it contains no free occurrences of variables.

The *order* of a term  $\tau$  is the order of its type.

The *degree* of  $\tau$  is the maximum order of its subterms.

We also define the *size* of a term inductively as follows. The size of a relational constant is the size of the corresponding instance, namely, the number of attributes times the number of rows. The size of a query constant is the size of its standard string representation – for each of the query constants above, the length should be clear; for example the size of a named-based projection  $\pi_A$  is the length of the string needed to represent all names in  $A$ , while for positional projection it is the size of a string that represents all positions or a position range. The *size of a variable* is the size of a standard string representation of the type of the variable. The size of a higher-order term is inductively defined as 1 plus the sum of the sizes of its top-level subterms.

EXAMPLE 2. Let  $R, R'$  be two variables of relational type  $\mathcal{R} = (a, b)$ , and  $Q$  be a variable of query type  $\mathcal{R} \rightarrow \mathcal{R}$ . We consider an order 1 degree 2 term below.

$$\tau_1 = (\lambda Q. \lambda R. Q(Q(R))) (\pi_{\{a,b\}} (\lambda R'. (\rho_{b/c}(R') \bowtie \rho_{a/c}(R'))))$$

We can see that  $\tau_1$  is of type  $\mathcal{R} \rightarrow \mathcal{R}$  and that it returns all the pairs of nodes having a path of length 4 between them. In general, using variables of order 2, we can find paths of length double exponential in the input size.

### Semantics.

Our semantics is simply the standard denotational semantics of the  $\lambda$ -calculus with interpreted constants. Throughout the definition below, we fix an interpretation for the constants in the signature  $\mathcal{F}$  as a function  $\mathcal{I}$  that maps every constant  $\text{const} \in \mathcal{F}$  to its semantics  $\llbracket \text{const} \rrbracket_{\mathcal{I}}$ . The semantic function  $\llbracket \tau \rrbracket_{\mathcal{I}}$  is defined with respect to an extension of the base interpretation to an interpretation that maps each free variable in  $\tau$  to an object of the corresponding type. The function is defined by induction for terms  $\tau$  in which each variable is abstracted only once, and is extended to all terms by performing renaming in the standard way.

- For every term  $\tau$  of the form  $\text{const}(\tau_1, \dots, \tau_k)$ ,  $\llbracket \tau \rrbracket_{\mathcal{I}}$  is defined as  $\llbracket \text{const} \rrbracket_{\mathcal{I}}(\llbracket \tau_1 \rrbracket_{\mathcal{I}}, \dots, \llbracket \tau_k \rrbracket_{\mathcal{I}})$ .
- Given a term  $\tau$  of the form  $\lambda X. \rho(X)$ ,  $\llbracket \tau \rrbracket_{\mathcal{I}}$  is the function that maps every object  $o$  in the type of  $X$  to the object  $\llbracket \rho \rrbracket_{\mathcal{I}[X/o]}$ , where  $\mathcal{I}[X/o]$  extends  $\mathcal{I}$  by assigning  $X$  to  $o$ .
- For  $\tau$  of the form  $\tau_1(\tau_2)$ ,  $\llbracket \tau \rrbracket_{\mathcal{I}}$  is the object  $\llbracket \tau_1 \rrbracket_{\mathcal{I}}(\llbracket \tau_2 \rrbracket_{\mathcal{I}})$ .

We will generally omit the base interpretation for constants in the remainder of the paper, and just talk about the evaluation of a term.

In the case where we have constants of order at most 1, this denotational semantics is equivalent to an operational semantics based on iteratively applying standard  $\lambda$ -reduction, along with the “built-in” semantics for a term consisting of a relational operator applied to relational constants. We omit the formalization of this, since an operational semantics will be implicit in the evaluation algorithms that witness our subsequent upper bounds.

We let  $\lambda_{DB}[\mathcal{F}]$  denote the term language built over constants in signature  $\mathcal{F}$ . For example  $\lambda_{DB}[\text{RA}^+]$  denotes the language built over data instance constants plus positive relational operators.

### Weakly-typed terms.

In the higher-order language defined above, we give types for all the variables and constants in terms. The maximal number of attributes of each term's type is polynomial in the term's size, since the type must be built in to each operator.

We now introduce a weakly-typed language which allows us to build terms that have types with exponential number of attributes in their size.

The syntax of weakly-typed terms is defined in the same way as the syntax of simply-typed terms except that we do not give types for variables. We only give explicit types for constants.

*Weakly-typed terms* are build up from constants in  $\mathcal{F}$  and variables in  $\mathcal{X}$  by using the operations of abstraction and application: every constant is a term of the constant's type; if  $X$  is a variable and  $\rho$  is a term, then  $\lambda X. \rho$  is a term; if  $\tau$  and  $\rho$  are two terms, then  $\tau(\rho)$  is a term.

A *consistent typing* of such a term is any assignment of types to subterms that is consistent with the typing of the data constants and the form of query operators; e.g. every instance of  $\times$  must have type  $T \rightarrow (T' \rightarrow T''')$  for some relational types  $T, T', T''$ , and consistent with function application and  $\lambda$  abstraction; e.g. if the type of  $\rho$  is  $T$  and the type of  $\tau(\rho)$  is  $T'$ , then the type of  $\tau$  must be  $T \rightarrow T'$ .

Later we will show that these conditions uniquely define a typing when one exists, and hence we can define the semantics of any typeable weakly-typed term to be that of the unique simply-typed term that is equivalent to it.

We only consider weakly-typed terms over the signature  $\text{RA}_x$ , and we denote the resulting language by  $\lambda_{DB}^-[ \text{RA}_x ]$ .

To provide intuition for weakly-typed terms, we consider the example below.

**EXAMPLE 3.** *Let  $D_0$  be a relational constant with two integer attributes and  $R_1, R_2$  be two relational variables. We consider a weakly-typed order 0 degree 1 term below.*

$$\tau_2 = \pi_{\{0,1\}} ((\lambda \mathbf{R}_2. (\mathbf{R}_2 \times \mathbf{R}_2)) (\lambda \mathbf{R}_1. (R_1 \times R_1) D_0))$$

*We infer types for variables  $R_1$  and  $R_2$  as follows. Since  $R_1$  is substituted by  $D_0$ , its type is the same as the type of  $D_0$ , consisting of two integer attributes. This also implies that the type of  $R_2$ , which is the same as the type of the subterm  $(\lambda R_1. (R_1 \times R_1) D_0)$ , has four integer attributes.*

Note that a consistent typing must give a single type to every variable.

### The evaluation problem.

We now define the main problem considered in this paper.

The *boolean query evaluation problem* takes as input a well-typed term of boolean query type, along with a set of relation constants, (i.e. relational instances). The output is true iff the application of the term on the instances evaluates to the (unique) nonempty instance of  $\{\}$ .

Of course, there are more general evaluation problems, which we will need as subproblems. We will be interested

in the *combined complexity* of the problem, where the size of the input is the database size plus the term size, as well as the *query complexity*, in which the database instance is fixed. It is easy to see that the complexity when the *query* is fixed is in polynomial time for all of our languages.

Note that we consider complexity in the standard Turing Machine model, and that we do not have any requirement on the behavior of our evaluation function when terms are not well-typed. We will consider the evaluation problem for both simply-typed terms and weakly-typed terms.

### Construction trees.

Tree and graph representations of  $\lambda$ -terms are commonly-used in analyzing reduction strategies for  $\lambda$  calculus. We will use a parsed representation that we call *construction trees* for our terms. The presence of constants requires a slight variation on prior representations (e.g [Wad71]).

Construction trees are defined only for terms whose constants have at most two inputs, e.g. relational constants, query operator constants. We also assume that query constants  $c$  always appear in subterms of the form  $\lambda R. c(R)$  or  $\lambda R. \lambda S. c(R, S)$ . For a binary constant  $c$  such as  $\cup, \bowtie, \times$ , we often use  $sct$  as a notation for  $c(st)$ .

The *construction tree* of a term  $\tau$  is a binary tree. Each application is represented by an @ node which has two subtrees. Each  $\lambda X$  is represented by a node, called a  $\lambda$  node. Each constant or variable is represented by a node labelled with its name.

The tree is inductively built as follows.

1. The root of the tree is the outermost operator.
2. The construction tree of a database constant  $c$  (resp. a variable  $x$ ) is a single leaf, labelled  $c$  (resp.  $x$ ).
3. The construction tree of an abstraction  $\lambda x. s$  consists of a node labelled  $\lambda x$  with a single subtree, which is the construction tree of  $s$ .
4. The construction tree of an application  $s(t)$  consists of a node labelled @ with two subtrees: the left subtree is the construction tree of  $s$  and the right subtree is the construction tree of  $t$ .
5. The construction tree of  $c(st)$  with  $c$  a binary query constant consists of  $c$  node and two subtrees: the left subtree is the construction tree of  $s$  and the right subtree is the construction tree of  $t$ .
6. The construction tree of a unary query constant  $c(s)$  consists of a node  $c$  whose only subtree is the construction tree of  $s$ .

### Query languages with inductive rules .

Our results will rely on reductions to several query languages with inductive definitions, most of them variants of Datalog.

An *atom* over a relational signature  $S$  is an expression  $R(x_1 \dots x_n)$  where  $R$  is an  $n$ -ary predicate of  $S$  and the  $x_i$  are either variables or constants. A *pure atom* is one in which all  $x_i$  are variables.

A positive rule block consists of:

- A relational signature  $S$  along with a collection of constants  $C$
- a set of rules of the form  $A \leftarrow \phi$ , where  $\phi$  is a conjunction of atoms over  $S$ , and  $A$  is an atom over  $S$ .  $A$  is the *head* of the rule and  $\phi$  is the *body* of the rule. We will often identify  $\phi$  with the set of atomic formulas in it, writing  $A_1(\vec{x}_1), \dots, A_n(\vec{x}_n)$  instead of  $A_1(\vec{x}_1) \wedge \dots \wedge A_n(\vec{x}_n)$ . A variable that occurs in the head of rule  $r$  is a *free variable* of  $r$ . Other variables are *bound* in  $r$ ; we write  $bvars(r)$  for the bound variables of  $r$ . We require that every free variable occurs in the body.
- A distinguished predicate  $P$  of  $S$  which occurs in the head of a rule, referred to as the *goal predicate*.

The relational symbols that do not occur in the head of any rule are the *input predicates*, while the others are *intensional predicates*. A predicate  $P$  immediately depends on another predicate  $P'$  if there is a rule that has  $P$  in the head and  $P'$  in the body. A rule block is *nonrecursive* if this relation is acyclic. We let NRDL denote the language of nonrecursive rule blocks.

All NRDL queries that we deal with here will be *pure*: that is all atoms in the heads of rules with nonempty bodies are pure.

We also consider the language of nonrecursive Datalog with negation, abbreviated NRDL $^\neg$ . This extends the prior syntax by allowing negation in rule bodies.

For a NRDL $^\neg$  query, we can define the *rank* of an intensional predicate  $P$  (with respect to the query, although we omit the argument), denoted  $\text{Rk}(P)$ , as follows: the rank of an input predicate is 0, the rank of an intensional predicate  $P$  is

$$\text{Rk}(P) = \max\{\text{Rk}(P') + 1 : \begin{array}{l} \text{there is a rule with } P' \text{ in} \\ \text{the body and } P \text{ in the head} \end{array}\}$$

Given a structure  $D$  interpreting the input predicates, an NRDL query  $Q$ , and a predicate  $P$  of  $Q$ , we define the evaluation of  $P$  in  $D$ , denoted  $P(D)$ , by induction on  $\text{Rk}(P)$ . For  $P$  an input predicate  $P(D)$  is the interpretation of  $P$  in  $D$ . For  $P$  an intensional predicate with  $\text{Rk}(P) = k + 1$  and arity  $l$ .

- Let  $D^k$  be the expansion of  $D$  with  $P'(D)$  for all intensional  $P'$  of rank at most  $k$ .
- If  $r$  is a rule with  $P(x_1 \dots x_l)$  in the head,  $\vec{w}$  the bound variables of  $r$ , and  $\phi(\vec{x}, \vec{w})$  the body of  $r$  let  $P_r(D)$  be defined by:

$$\{\vec{c} \in \text{Dom}(D)^l : (D^k, x_1 \mapsto c_1 \dots x_l \mapsto c_l) \models \exists \vec{w} \phi\}$$

We let  $P(D)$  denote the union of  $P_r(D)$  over all  $r$  with  $P$  in the head. The *result* of a query  $Q$  on  $D$  is the evaluation of the goal predicate of  $Q$  on  $D$ .

Finally, the language of *Inflationary Fixed Point logic* over relational signature  $S$ , is a collection of blocks of rules  $B_1 \dots B_j$ , where each  $B_i$  is an NRDL $^\neg$  query whose input predicates consist of  $n_i$  *recursive* predicates which can appear in both heads and bodies, and include all predicates in the head, along with  $p_i$  *parameter* predicates, which occur only in rule bodies. For the lowest block  $B_1$ , we must have

the parameter predicates being a subset of the input signature. For other blocks we must have the parameter predicates be a subset of the recursive predicates in the previous block. The top level block  $B_j$  has a distinguished recursive predicate, again called the goal. We evaluate a query by induction on  $j$ , getting instances for each recursive predicate in  $B_j$ . At stage  $i$ , we substitute for each parameter predicate in block  $B_i$  with the inductively computed output of  $B_{i-1}$ , or the input predicates if  $i = 1$ . We then iteratively compute values for the recursive predicates  $P_1 \dots P_{n_i}$  by repeating the following assignment until a fixpoint is reached:  $P_i := \{\vec{x} : (\exists \vec{y} \text{ body}(P_i)(\vec{x}, \vec{y})) \vee P_i(\vec{x})\}$ , where  $\text{body}(P_i)$  is the disjunction of all bodies of rules with  $P_i$  at the head, with the predicates  $P_1 \dots P_n$  replaced by the result of the prior iteration.

### 3. LOW DEGREE TERMS

We begin with the case of terms having only relation and query variables; most of the distinguishing factors of  $\lambda_{DB}$  from other higher-order calculi are contained in these cases.

#### *Degree 1 Terms.*

Naive evaluation of a degree 1 term would be to simply substitute all occurrences of a relational variable by their bodies, and then evaluate the resulting variable-free term using the fixed semantics of relational calculus. Unfortunately, this would involve an exponential blow up. Instead, we follow the approach of [BPV10], which reduces the problem to the evaluation for fragments of Datalog.

PROPOSITION 1. *Evaluation of degree 1 terms over RA is linearly inter-reducible to NRDL $^\neg$  evaluation, while for RA $^+$  we can reduce to and from NRDL evaluation.*

The following example demonstrates the reduction between degree 1 terms and Datalogqueries.

EXAMPLE 4. *Let  $R, R_1$  and  $R_2$  be three relational variables respectively of type  $(b, c), (a, b), (a, b, c)$ . We consider an order 1 degree 1 term below.*

$$\tau_3 = \lambda R_1. \lambda R_2. (\lambda \mathbf{R}. \pi_{\{a\}}((\mathbf{R}_1 \bowtie \sigma_{c=5}(\mathbf{R}))))(\pi_{\{b,c\}} R_2)$$

The term  $\tau_3$  is equivalent to the following NRDL query.

$$\begin{array}{l} \text{Goal}(x) \leftarrow R_1(x, y), R(y, 5) \\ R(y, z) \leftarrow R_2(x, y, z) \end{array}$$

where  $R_1, R_2$  are two input predicates and  $R$  is an intensional predicate.

This result gives us a PSPACE upper bound for the complexity of evaluating degree 1 terms over RA. The PSPACE upper bound is tight, even for terms with no negation and no union [Var82, DEGV01, BPV10], and even for query complexity.

We will need the following simple upper bound when evaluating NRDL $^\neg$ :

PROPOSITION 2. *The problem of evaluating an NRDL $^\neg$  query  $\mathcal{P}$  over database  $D_0$ , where  $\mathcal{P}$  has  $N$  rules of size at most  $L$ , can be done in  $O(N \times |D_0|^L)$  time.*

PROOF. Evaluate the query bottom-up. Since the size of each rule is bounded by  $L$ , the output of each rule is bounded by  $|D_0|^L$ , and each rule can be evaluated in  $O(|D_0|^L)$  time.  $\square$

### Degree 2 Terms.

For degree 2, we will make use of our results from degree 1, plus observations inspired by Schubert's work [Sch01] on the complexity of normalization for low-degree terms in the standard  $\lambda$ -calculus.

The following result gives the complexity of degree 2 terms in the strongly-typed languages  $\lambda_{DB}[\text{RA}]$  and  $\lambda_{DB}[\text{RA}^+]$ .

**THEOREM 3.** *The problem of evaluating degree 2 terms over either RA or  $\text{RA}^+$  is EXPTIME-complete.*

We begin with the upper bound, stating it only for the larger signature RA.

**PROPOSITION 4.** *The problem of evaluating degree 2 terms over RA is in EXPTIME.*

**PROOF.** We perform standard innermost-reduction to reduce a degree 2 term to a degree 1 term. In the process, the size of the term increases exponentially. However, we observe that the increase is only exponential, and that the arity of every order 1 output (including intermediate relations) does not increase. Hence we can evaluate the resulting NRDL<sup>-</sup> expressions in exponential time.  $\square$

**PROPOSITION 5.** *The problem of evaluating degree 2 terms is EXPTIME-hard, even for  $\lambda_{DB}[\text{RA}^+]$ .*

**PROOF.** We will use the union operator in the following proof, but we can eliminate it by using the standard method of encoding disjunction with additional arguments for intermediate truth values – a method presented in [GP03, VV98].

We show that the problem is EXPTIME-hard by reducing from the acceptance problem for an exponential time Deterministic Turing Machine (DTM)  $\mathcal{M}$  over an input  $\omega$  with  $|\omega| = n$ . The DTM  $\mathcal{M}$  is defined as a set  $(Q, \Sigma, \Gamma, \delta, q_0, \mathcal{F})$  with:

- $Q$ : a finite set of states,
- $\Sigma$ : input alphabet - a finite alphabet of symbols,
- $\Gamma \supseteq \Sigma \uplus \{\square\}$ : working tape alphabet - a finite alphabet of symbols,
- $\delta : (Q \setminus F) \times \Gamma \rightarrow Q \times \Gamma \times \{01, 00, 10\}$ : the transition function, where 01 denotes a rightward move of the head, 10 a leftward move, and 00 no movement of the head,
- $q_0 \in Q$ : the initial state,
- $\mathcal{F} \subseteq Q$ : a set of final states.

The DTM operates on an infinite tape, which is assumed to be bounded to the left. Each cell of the tape contains one symbol from  $\Gamma$  which contains  $\square$ , a blank symbol.

We now give a reduction from the acceptance problem of a DTM that runs in less than  $2^n$  steps to our evaluation problem. Since  $\delta$  is a partial function, we add  $\delta(q, a) = (q, a, 00)$  for all  $(q, a)$  such that  $\delta(q, a) = \emptyset$  to extend  $\delta$  to a total function. We consider only DTM's that run in exactly  $2^n$  steps. We will check the state of the DTM at step  $2^n$  to know if it accepts or rejects. With this extension of  $\delta$ , DTM's that halt after  $s$  steps with  $s < 2^n$  will stay at the same configuration from  $s + 1$  to  $2^n$ .

We use the following relations and queries over a database domain  $\{0, 1\}$ .

- We will use relations of the form  $S(p_1, \dots, p_n, a_1, \dots, a_k, h, b_1, \dots, b_m)$  (shortly,  $S(\vec{p}, \vec{a}, h, \vec{b})$ ) to represent the configuration of  $\mathcal{M}$  at a particular time. Each tuple represents a cell of the tape. Specifically, the attributes of  $S$  have the following roles:

- $\vec{p}$  represents the distance in binary of the cell to the left end of the tape.
- $\vec{a}$  represents the symbol on that cell in binary.
- $h$  is 1 if the head is on that cell; otherwise  $h$  is 0.
- $\vec{b}$  represents the state of  $\mathcal{M}$ .

- The succinctness of degree 2 terms is used to build a function  $\tau$  of size  $O(n)$  that takes a relation  $R$  of type  $\mathcal{T}$  and a query  $Q$  of type  $\mathcal{T}_Q = \mathcal{T} \rightarrow \mathcal{T}$ , returning  $Q^{2^n}(R)$ .

$$\tau = \tau_n = \lambda x_{n-1}. \lambda Q. \lambda R. x_{n-1}(Q, x_{n-1}(Q, R)) \tau_{n-1}$$

...

$$\tau_0 = \lambda Q. \lambda R. Q(R)$$

Each  $\tau_i$  is of type  $(\mathcal{T} \rightarrow \mathcal{T}) \rightarrow \mathcal{T} \rightarrow \mathcal{T}$  (i.e.  $\mathcal{T}_Q \rightarrow \mathcal{T}_Q$ ).

- $T(\vec{b}, \vec{a}, \vec{b}', \vec{a}', c_1, c_2)$  is a relation that stores all the transitions of  $\delta$ . The roles of  $\vec{a}, \vec{a}'$  and  $\vec{b}, \vec{b}'$  are the same as their roles in  $S$ . In addition,  $c_1$  and  $c_2$  tell us if the head moves to the right ( $c_1 c_2 = 01$ ), moves to the left ( $c_1 c_2 = 10$ ), or stays ( $c_1 c_2 = 00$ ).
- A relation  $F(\vec{b})$  is used to store all the final states in  $\mathcal{F}$ .
- We use two degree 1 terms to represent succ and diff over any relation  $P$  of type  $(p_1, \dots, p_n)$ . Both succ and diff are of type  $(p_1, \dots, p_n) \rightarrow (p_1, \dots, p_n, v_1, \dots, v_n)$ , which is also denoted by  $(\vec{p}) \rightarrow (\vec{p}, \vec{v})$ .

We define diff as follows.

$$\text{diff} = \lambda P. \bigcup_{1 \leq i \leq n} (\sigma_{p_i=0, v_i=1}(P^*) \cup \sigma_{p_i=1, v_i=0}(P^*))$$

with  $P^* = P \bowtie (\rho_{\vec{p}/\vec{v}}(P))$ .

The following term defines succ.

$$\text{succ} = \lambda P. \bigcup_{1 \leq i \leq n} \sigma_C (P \bowtie (\rho_{\vec{p}/\vec{v}}(P)))$$

with

$$C = (p_1 = v_1, \dots, p_{i-1} = v_{i-1}, p_i = 0, v_i = 1, p_{i+1} = 1, \dots, p_n = 1, v_{i+1} = 0, \dots, v_n = 0)$$

- Using diff and succ, we can define a term  $\rho_0$  of type  $(\vec{p}, \vec{a}, h, \vec{b}) \rightarrow (\vec{p}, \vec{a}, h, \vec{b})$  which represents the next configuration of  $\mathcal{M}$ . If  $S$  represents the current tape configuration,  $\rho_0(S)$  is the next configuration of  $\mathcal{M}$ . The term  $\rho_0$  is the union of the following three terms. The first term represents the new description of the cell which the head is on. The second one represents the new description of the cell to which the head will go. The third one represents the description of the cells which do not change in the transaction.

Let  $S_0$  be an instance of  $S$  that represents the input  $\omega$ . Then,  $\tau(\rho_0, S_0)$  will represent the state of the DTM  $\mathcal{M}$  after  $2^n$  steps.

We define  $\rho = \tau(\rho_0, S_0) \bowtie F$ . It is easy to see that  $\rho \neq \emptyset$  iff  $\mathcal{M}$  accepts  $\omega$  within  $2^n$  steps.  $\square$

**PROPOSITION 6.** *The problem of evaluating degree 2 terms remains EXPTIME-hard when we fix either the relational constants or the types of all variables and constants in the term.*

**PROOF.** First we consider the case where the relational constants are fixed. In the EXPTIME-hardness above, we only use relational constants  $S_0$  to encode the input tape  $\omega$ . We assume that the type of  $S_0$  is  $\{0, 1\}^m$  and  $S_0 = \{t_1, \dots, t_n\}$ . We use a single attribute relation instance  $D_0$  with values 0 and 1.

Now we can code  $S_0$  by the following expression:

$$S_0 = \bigcup_{1 \leq i \leq n} \left( \sigma_{\vec{A}=t_i}(\rho_{A/A_1}(D_0) \bowtie \dots \bowtie \rho_{A/A_m}(D_0)) \right)$$

where  $\vec{A} = \{A_1, \dots, A_m\}$  is a set of different attribute names of integer type.

When the types are fixed, we use queries to represent elements of  $\mathcal{D}^1$ . A binary relation instance  $Next = \{\langle 1, 2 \rangle, \langle 2, 3 \rangle, \dots, \langle n-1, n \rangle\}$  stores an ordered set of  $n$  elements. The query variables of type mapping from the domain  $\{1, \dots, n\}$  to the domain  $\{0, 1\}$  will range over a domain of  $2^n$  size. We also can easily represent basic ordering operators on this domain. Thus we still can code an ordered set which has an exponential number of elements. The other steps are similar to the EXPTIME-hardness proof above.  $\square$

### Adding recursion.

We now study the effect of the order 2 constant, **ifp**, to the evaluation problem of low degree terms.

The proposition below shows that **ifp** evaluation can be reduced to evaluation with terms of degree 2 and only the standard relational query constants. It gives an alternative proof of Proposition 5, as explained below.

**PROPOSITION 7.** *Evaluation of a degree 2 term in  $\lambda_{DB}[\text{RA}]$  or in  $\lambda_{DB}[\text{RA}^+]$  containing **ifp** is polynomially reducible to evaluation of a degree 2 term over the same signature without **ifp**.*

**PROOF.** Let  $n$  be the size of the relational constants and the term. Since the size of any intermediate relational instance formed is bounded by  $O(2^n)$ , we can calculate the least fixed point value **ifp**( $Q, D_0$ ) for any query  $Q$  and fixed  $D_0$  of size at most  $n$  by  $Q'^{2^n}(D_0)$ , where  $Q'$  is the modification of  $Q$  to union with its input. Moreover, there is a small degree 2 term that transforms any query  $Q$  of a given query type to  $Q'$  and there is a degree 2 term of size  $O(n)$  that iterates any given  $Q$  on any given  $R$  (depending only on the types of  $Q$  and  $R$ )  $2^n$  times – as shown in Proposition 5. Thus we can transform a term formed from **ifp** by replacing subterms **ifp**( $\rho, \rho'$ ) with a degree 2 term applied to  $\rho$  and  $\rho'$ . Doing this iteratively gives the desired transformation.  $\square$

The result above implies that the complexity of evaluating terms of degree from 2 does not change when **ifp** is included

– hence it is EXPTIME-complete.

From well-known results on query languages with recursion, we see that the **ifp** constant does have some impact on the complexity of degree 1 evaluation.

**PROPOSITION 8.** *Evaluation of a degree 1 term in  $\lambda_{DB}[\text{RA}^+]$  or in  $\lambda_{DB}[\text{RA}]$  containing **ifp** is EXPTIME-complete.*

The upper bound is inherited from the degree 2 case. Hardness follows from the EXPTIME-hardness of Datalog in query complexity [DEGV01], and the fact that Datalog can easily be embedded in **ifp**.

## 4. ARBITRARY DEGREE TERMS

The main purpose of this section is to show the following theorem.

**THEOREM 9.** *The problem of evaluating degree  $k$  terms is:*

- $m$ -EXPTIME-complete if  $k = 2m$ , i.e.  $k$  is even,
- $m$ -EXPSPACE-complete if  $k = 2m + 1$ , i.e.  $k$  is odd.

Note: 0-EXPSPACE denotes PSPACE and 0-EXPTIME denotes PTIME.

### 4.1 Upper bounds for evaluating degree $k$ terms

Let  $\text{exp}_m^n$  be 2 to the  $m$ -hyperexponential of  $n$ , i.e.  $2^{2^{\cdot^{2^n}}}$  with a tower of  $m$  2's.  $m$ -EXPTIME below refers to the class of functions that run in time  $\text{exp}_m^{n_{O(1)}}$ , and similarly for  $m$ -EXPSPACE. Based on the techniques in [HK96], we give the following upper bound.

**PROPOSITION 10.** *The problem of evaluating degree  $k$  terms is:*

- in  $m$ -EXPTIME if  $k = 2m$ ,
- in  $m$ -EXPSPACE if  $k = 2m + 1$ .

**PROOF.** • We show that the problem of evaluating degree  $k$  terms with  $k = 2m$  and  $m \geq 0$  is in  $m$ -EXPTIME.

We use  $m$ -hyperexponential time to reduce a degree  $2m$  term  $\tau$  to a degree  $m$  term  $\tau'$ . Similarly to the EXPTIME evaluation of degree 2 terms, we can evaluate  $\tau'$  in  $m$ -hyperexponential time: simply reduce to degree 1, blowing up the size but not the arity, and then apply the evaluation strategy for degree 1. Thus, the problem is in  $m$ -EXPTIME.

- Now we show that the problem of evaluating degree  $k$  terms with  $k = 2m + 1$  and  $m \geq 0$  is in  $m$ -EXPSPACE.

We use  $m$ -hyperexponential time to reduce a degree  $(2m + 1)$  term  $\tau$  to a degree  $(m + 1)$  term  $\tau'$ . Since  $\tau'$  is of degree  $(m + 1)$ , its variables are of order at most  $m$  which can be non-deterministically guessed using  $m$ -hyperexponential space. Since  $\tau'$  is  $m$ -hyperexponential in the size of  $\tau$ , we can use  $m$ -hyperexponential space to represent guesses of all the variables in  $\tau'$ . Then we can also evaluate  $\tau'$  in  $m$ -hyperexponential space by a top-down algorithm.

$\square$

The argument here relies on reduction to the order 2 case via  $\beta$ -reduction, plus a few properties of the constants. The same argument is easily seen to hold for the extension with recursion.

## 4.2 Coding ordered sets by degree $k$ terms

Before showing the lower bound, we describe how to use degree  $k$  terms to code a  $k$ -hyperexponential set.

Let  $X$  be a term of type  $T^0$  and  $Y$  be a term of type  $T^0 \rightarrow T^0$ . We define *iteration functions* that return  $Y^{\text{exp}_k^n}$  when applying over  $X$ .

**PROPOSITION 11.** *Given  $X$  of type  $T^0$  and of order  $k_0$  and  $Y$  of type  $T^1 = T^0 \rightarrow T^0$ , there exists a term of degree  $k_0 + k$  that returns  $Y^{\text{exp}_k^n}(X)$ .*

**PROOF.** We build these terms as follows.

| Subterm    |  |
|------------|--|
| $\tau_n^2$ | $= \lambda x_{n-1}^2 . \lambda x^1 . \lambda x^0 . x_{n-1}^2 (x^1, x_{n-1}^2 (x^1, x^0)) \tau_{n-1}^2$   |
| $\dots$    |  |
| $\tau_0^2$ | $= \lambda x^1 . \lambda x^0 . x^1 (x^0)$  |
| $\dots$    |  |
| $\tau_n^i$ | $= \lambda x_{n-1}^i . \lambda x^{i-1} \dots \lambda x^0 .$<br>$x_{n-1}^i (x^{i-1}, \dots, x^1, x_{n-1}^i (x^{i-1}, \dots, x^0)) \tau_{n-1}^i$ |
| $\dots$    |  |
| $\tau_0^i$ | $= \lambda x^{i-1} \dots \lambda x^0 . x^{i-1} (x^{i-2}, \dots, x^0)$  |

where each subterm  $\tau_j^i$  with  $2 \leq i \leq k$  and  $0 \leq j \leq n$  has type  $T^i$  defined as below:

$$\begin{aligned} T^2 &= T^1 \rightarrow T^1 = T^1 \rightarrow T^0 \rightarrow T^0 \\ T^i &= T^{i-1} \rightarrow T^{i-1} = T^{i-1} \rightarrow \dots \rightarrow T^0 \rightarrow T^0 \end{aligned}$$

From that we define a term as follows.

$$\rho_n^k = \lambda Y . \lambda X . (((\tau_n^k \tau_n^{k-1}) \dots \tau_n^2) Y) X$$

This term takes two terms  $X, Y$  and returns  $Y^{\text{exp}_k^n}(X)$ .  $\square$

These iterators will be used to capture both a large amount of space and a large amount of time: we will use them to scan through a large domain, allowing us to code a huge tape. Later on we will use them to iterate a state transition function a large number of times. We start with the first application, using iterators to code a set of  $k$ -hyperexponential elements. Actually, we do not code all the elements. We only code one element and provide a term that generates all the other ones.

**PROPOSITION 12.** *We can efficiently construct an order  $m-2$  term  $\text{Cell}_{m-2}$  over  $\text{RA}^+$  of some type  $\Delta_{m-2}$  and an order  $m-1$  term  $\text{succ}_{m-2}$  of type  $\Delta_{m-2} \rightarrow \Delta_{m-2}$  such that by iterating  $\text{succ}_{m-2}$  on  $\text{Cell}_{m-2}$ , we get a set  $\mathcal{S}_{m-2}$  of objects. We also can build boolean equality and inequality functions  $=_{m-2}$  and  $\text{diff}_{m-2}$  to compare the objects in  $\mathcal{S}_{m-2}$ . With respect to the semantics of  $\text{diff}_{m-2}$ ,  $\mathcal{S}_{m-2}$  contains  $\text{exp}_m^n$  distinct objects.*

**PROOF. Base case ( $m=2$ ):** Let  $D_1, \dots, D_{n+1}$  be  $n+1$  single attribute relational constants respectively of type  $(a_1), \dots, (a_n), b$ . Each  $D_i$  contains two tuples  $\langle 0 \rangle$  and  $\langle 1 \rangle$ , i.e.  $D_i = \{\langle 0 \rangle, \langle 1 \rangle\}$  with  $1 \leq i \leq n+1$ . We build up an instance  $D_0$  using the term  $D_1 \boxtimes \dots \boxtimes D_{n+1}$ .

Now  $D_0$  contains every tuple of  $n+1$  attributes over  $\{0, 1\}$ . We will be interested in instances for which there is exactly one tuple that projects onto every combination of boolean values for the first  $n$  attributes. Such an instance will represent a function from  $2^n$  to 2, hence a number bounded by  $2^{2^n}$ . The initial position  $\text{Cell}_0$  satisfies the last attributes of all the  $2^n$  tuples equivalent to 0. We now describe  $\text{succ}_0$ . Consider a number in  $2^{2^n}$  as a sequence of  $2^n$  bits.  $\text{succ}_0$  should find the first bit that is 0 and flip it to 1. Then we build  $=_0$  and  $\text{diff}_0$ .

**Induction case:** We now show how to inductively define  $\text{Cell}_{k+1}$ ,  $\text{succ}_{k+1}$ ,  $=_{k+1}$ , and  $\text{diff}_{k+1}$  from  $\text{Cell}_k$ ,  $\text{succ}_k$ ,  $=_k$ , and  $\text{diff}_k$ . We define  $\text{Cell}_{k+1}$  as a degree  $k+1$  term of type  $\Delta_k \rightarrow \mathcal{T}_0$  with  $\mathcal{T}_0 = (u)$ . Intuitively,  $\text{Cell}_{k+1}$  always returns  $\{\langle 0 \rangle\}$  when applying to an element of the set  $\{\text{Cell}_k, \text{succ}_k(\text{Cell}_k), \dots, (\text{succ}_k)^{\text{exp}_k^n}(\text{Cell}_k)\}$ . Using iterators from Proposition 11, we define  $\text{succ}_{k+1}$ ,  $=_{k+1}$ , and  $\text{diff}_{k+1}$ .  $\square$

## 4.3 Lower bounds for degree $k$ terms

Using Proposition 11, Proposition 12 and the techniques inspired by [HK96, Mai92], we can show that the preceding upper bounds are tight.

**PROPOSITION 13.** *The problem of evaluating degree  $k$  terms is:*

- $m$ -EXPTIME-hard if  $k = 2m$ ,
- $m$ -EXPSPACE-hard if  $k = 2m + 1$ .

**PROOF.** We give the intuition of the proof for both parts of the proposition.

- To show that the problem of evaluating degree  $k$  terms with  $k = 2m$  is  $m$ -EXPTIME-hard, we reduce the satisfiability of an  $m$ -hyperexponential time DTM over an input of  $m$ -hyperexponential size to this problem. We use a set of order  $m-1$  terms to represent configurations of  $m$ -hyperexponential size. Then we use an order  $m$  term to simulate transitions from a configuration to the next configuration. Lastly, we use a term of degree  $2m$  to repeat that order  $m$  term  $m$ -hyperexponential times.
- To show that the problem of evaluating degree  $k$  terms with  $k = 2m + 1$  is  $m$ -EXPSPACE-hard, we reduce the satisfiability of an  $m$ -hyperexponential space DTM over an input of  $m$ -hyperexponential size to this problem. We use a set of order  $m-1$  terms to represent configurations of  $m$ -hyperexponential size. Then we use an order  $m$  term to simulate transitions from a configuration to the next configuration. Lastly, we use a term of degree  $2m + 1$  to repeat that order  $m$  term  $m$ -hyperexponential times.

$\square$

Similarly to Proposition 6, we can show that the complexity of evaluation does not change in some particular cases.

**PROPOSITION 14.** *When we fix either the relational constants or the arities of all relational types occurring in the term the hardness results in Proposition 13 do not change.*

The results above imply that we can not find an integer number  $N$  such that the evaluation problem is in  $N$ -EXPTIME. That is:

COROLLARY 15. *The evaluation problem for higher-order terms has non-elementary complexity.*

## 5. REDUCING THE COMPLEXITY OF EVALUATION

We consider particular cases where we can achieve better bounds for the complexity of evaluation.

### Linear higher-order terms.

Linear Datalog queries (see, e.g. [DEGV01]) disallow repeated occurrences of an intensional predicate in a rule body. The following generalizes this to terms of arbitrary degree.

A closed term  $\tau$  is *linear* iff  $\tau$  does not contain two occurrences of the same variable. The following result is clear:

THEOREM 16. *A linear term  $\tau \in \lambda_{DB}[\mathcal{F}]$  can be transformed in linear time to an equivalent expression in the term algebra over  $\mathcal{F}$ , using standard  $\beta$ -reduction.*

PROOF. We can reduce  $\tau$  to an equivalent degree 0 term  $\tau'$  tractably via substitution. By the definition of linear terms, the size of  $\tau'$  is the same as the size of  $\tau$ .  $\square$

Thus, evaluating a linear term  $\tau \in \lambda_{DB}[\mathcal{F}]$  has the same complexity as evaluating an expression in  $\mathcal{F}$ , which is NP-complete for  $\text{RA}^+$  and PSPACE-complete for RA.

### Un-nested terms.

In linear terms we never repeat variables. However, we notice that the source of non-elementary hardness in evaluation in the previous section comes from the ability to repeat variables in a particular way: nesting one occurrence of a variable inside another. By means of construction trees, we define a restricted class of terms in which the nesting of functions is limited. A variable  $x \in t$  is *self-nested* if  $x$  occurs in two subtrees  $s, t$  of the construction tree of  $t$  and two roots of  $s$  and  $t$  are linked to the same @ node.

A term is un-nested if it does not contain any self-nested variable. Intuitively, a term is un-nested if a variable never occurs as an argument of itself in the term. For example,  $\lambda QD.Q(Q(D))$  has a self-nested variable, but  $\lambda QD_1D_2.Q(D_1) \bowtie Q(D_2)$  is un-nested.

We show that the un-nested case is simpler, using two reductions. In the first, we reduce un-nested terms to degree 1 terms using only polynomial space.

PROPOSITION 17. *There is a PSPACE algorithm that reduces a degree  $k$  term  $\tau^k$  of size  $n$  to a degree 1 term  $\tau^1$ . The height of the construction tree of  $\tau^1$  is bounded by  $O(n)$ .*

PROOF. Given a degree  $k$  term  $\tau^k$  and its construction tree  $\xi$ , we go top-down through  $\xi$  to find pairs of a variable  $X$  and the subtree  $T$  to which the variable maps. We store those pairs  $(X, T)$  in a list named  $\mathcal{L}$ . This produces a function  $\mathcal{L}$ , which will not change in the recursive process defined below.

Now we are ready to give the algorithm, **Reduction** which takes as input a node  $C$  in a tree  $T$  and returns a new term:

- If  $C$  is a constant of arity 2 (e.g.  $\cup$ ) or an @ node with a right-child of order 0. Let  $C_l$  and  $C_r$  be the

left and right children respectively. In this case return a tree rooted at  $C$  with subtrees **Reduction**( $C_l, T$ ) and **Reduction**( $C_r, T$ ). If  $C$  is a unary operator with child  $C'$  we return a tree rooted at  $C$  with single child **Reduction**( $C', T$ ).

- If  $C$  is an @ node with a right-child of order  $\geq 1$ , then its left child must be of the form  $\lambda X$  with a child  $C''$  whose type matches the type of the right-child; in this case return **Reduction**( $C'', T$ ).
- If  $C$  is a variable node  $X$  of order from 2 then return **Reduction**( $\text{root}(\mathcal{L}(X)), \mathcal{L}(X)$ ).
- If  $T$  does not have variables of order above 1 then return  $T$ .

The initial call is **Reduction**( $\text{root}(\xi), \xi$ ). The output of **Reduction**( $\text{root}(\xi), \xi$ ) is the construction tree of an equivalent term  $\tau^1$  without variables of order more than 1. An implementation needs to explore only one branch at a time. Thus the algorithm requires polynomial space. During each call to **Reduction**, we only replace a variable at most once by a subtree of the original tree. Thus the height of the output construction tree is bounded by  $O(n)$ .  $\square$

From the proposition above, we can use a PSPACE algorithm to reduce an un-nested term of size  $n$  to an  $\text{NRDL}^\neg$  query  $\mathbf{Q}$ . The rank of the predicates in  $\mathbf{Q}$  is bounded by  $O(n)$  because the height of the construction tree is bounded. Since during the reduction we do not combine query constants, the length of the rules is still bounded by  $n$ . We now show that this  $\text{NRDL}^\neg$  query can be evaluated using  $O(n)$  space.

PROPOSITION 18. *There is an evaluation algorithm for  $\text{NRDL}^\neg$  queries which uses space polynomial in the maximal rank of the predicates and the maximal length of the rules.*

The proof uses the standard top-down algorithm for evaluating  $\text{NRDL}^\neg$  queries.

The two propositions above directly imply the following result.

THEOREM 19. *The evaluation problem for un-nested terms in  $\lambda_{DB}[\text{RA}]$  is in PSPACE.*

The same argument reducing to the degree 1 case can be used for **ifp**, yielding the following:

PROPOSITION 20. *The evaluation problem for un-nested terms in  $\lambda_{DB}[\text{RA}]$  containing **ifp** is in EXPTIME.*

### Fixing parameters in the problem.

We look at several parameters in the evaluation problem: for example fixing the relational constants and the size of the variables, which is the size of standard string representations of the types of the variables.

To reduce the complexity of evaluating degree 1 terms, it is sufficient to fix only the arities of variables.

PROPOSITION 21. *When we fix the arities of the relational variables, the evaluation problem for degree 1 terms is NP-complete for  $\text{RA}^+$  and RA.*

The proof of the upper bound is simply via “bottom-up” evaluation of the corresponding  $\text{NRDL}^\neg$  rules. The lower bound follows since we can still code conjunctive queries of arbitrary size.

However, to reduce the complexity of degree above 1 terms, we need to fix both the relational constants and the size of the variables.

**PROPOSITION 22.** *When we fix all the relational constants and the size of all the variables, the evaluation problem for degree 2 terms is NP-complete for  $\text{RA}^+$ , and PSPACE-complete for RA.*

**PROOF.** NP-hardness is obvious from Proposition 21, and similarly PSPACE-hardness for RA follows from classical results. So we show membership.

Let  $D$  be the set of values that appear in the term, including in the input data. For any type  $\mathcal{T}$ , let  $\mathcal{T} \upharpoonright D$  be the restriction of  $\mathcal{T}$  to values in  $D$ , defined inductively. For a relational type  $\mathcal{T} = (a_1 \dots a_n)$ , it is the type obtained by replacing the range of each  $a_i$  with the intersection of its range with  $D$ .  $(\mathcal{T} \rightarrow \mathcal{T}') \upharpoonright D$  is  $(\mathcal{T} \upharpoonright D) \rightarrow (\mathcal{T}' \upharpoonright D)$ . Since  $D$  is finite, each  $\mathcal{T} \upharpoonright D$  has only finitely many elements. Once the arity of base relations and the number of arguments to query types are fixed, the number of elements is uniformly bounded and independent of the type. Similarly, for any term  $\tau$  of type  $\mathcal{T}$  we define its restriction  $\tau \upharpoonright D$ , which will be of type  $\mathcal{T} \upharpoonright D$ .

An *extension guess* for order 0 term  $\tau$  is a mapping taking every variable of  $\tau$  of type  $T$  to an object of type  $T \upharpoonright D$ . A guess  $g$  is extended to all subterms of  $\tau$  by setting  $g(\tau_1(\tau_2)) = g(\tau_1)(g(\tau_2))$  and  $g(\lambda Q\tau) = g(\tau)$ , and propagating through relational operators homomorphically. This extension can be done in NP for  $\text{RA}^+$ , since it requires just evaluating the relational operators. Similarly it can be done in PSPACE for RA.

A guess  $g$  is correct if for every subterm of the form  $(\lambda R.\tau_1)(\tau_2)$  we have  $g(R) = g(\tau_2)$ , and similarly for query variables. It is easy to see that a correct guess must map  $\tau$  to its evaluation, and that there is a correct guess for every term.

Our algorithm exhaustively checks all extension guesses for correctness until it finds a correct one (which it must, by the above). The number of guesses is fixed, and checking for correctness requires calculating the extension and checking the equality of every such  $R$  and  $\tau_2$ . Calculating the extension is in the required class, as noted above. A correctness check requires only linearly many checks of variables. Since the possible values of variables are fixed, each equivalence check can be done in constant time.

□

The following proposition is easily generalized from the proposition above.

**PROPOSITION 23.** *When we fix all the relational constants and the size of all the variables, the evaluation problem is NP-complete for  $\text{RA}^+$ , and in PSPACE for RA (hence PSPACE-complete for RA when the degree is at least 2).*

Given that each of our languages subsume conjunctive queries, NP combined complexity is a reasonable goal. Our focus on combined and query complexity is justified by the fact that once a term of query type is fixed, the complexity of

normalization is fixed, and hence the evaluation complexity is the same as the data complexity of the corresponding language of ground terms over the signature (which will be just the data complexity of  $\text{NRDL}$ ,  $\text{NRDL}^\neg$ ,  $\text{ifp}$ , respectively). Hence:

**PROPOSITION 24.** *The evaluation problem for fixed terms of query type for any of our signatures can be done in polynomial time.*

## 6. EVALUATING WEAKLY-TYPED DEGREE 1 TERMS

Our upper bounds in degree 2 relied heavily on the fact that we could not build up terms of high arity, and in particular we could only build up relational instances of size exponential in the input. This is in contrast with XML query languages like XQuery, which allows the formation of doubly-exponential sized outputs.

We now study evaluation for the weakly-typed languages, which do allow the building of doubly-exponential sized objects. We will be concerned with the evaluation problem only for terms that have a consistent typing. Although there can be multiple consistent typings for a term (e.g. consider  $\lambda S.D_0$  for  $D_0$  a relational constant) it is easy to show that the evaluation of order 0 terms does not depend on the typing. Nevertheless, we will need some results on the complexity of the typing problem as the basis of our algorithms.

First we consider the typing problem for weakly-typed degree 1 terms.

**PROPOSITION 25.** *There exists a PTIME algorithm that takes a weakly-typed degree 1, order 0 term and determines whether it has a consistent typing, returning a typing if it does have one.*

**PROOF.** We give an algorithm working on the construction tree of the term that give types for all subterms or reports that there is no consistent type. The algorithm iterates the following process until all subterms are typed or inconsistency is detected:

- If we find a subterm rooted at a constant where all children have types, then do the following: If the constant is incompatible with the typing of its children (e.g. a selection selects an index that is bigger than the arity of the term it selects from), then inconsistency is detected and the algorithm terminates. Otherwise the typing is propagated to the subterm in the obvious way (e.g. if  $\tau_1$  has type  $m$  and  $\tau_2$  type  $n$ ,  $\tau_1 X \tau_2$  has type  $m + n$ ).
- For a subterm of the form  $(\lambda R\tau)\tau'$ , if we have a type for  $\tau'$  we propagate it to  $R$ .

Since one additional subterm will be typed in every iteration, the algorithm terminates in PTIME. □

One can also see that any typing must satisfy the inductive rule given by the algorithm, hence:

**PROPOSITION 26.** *There is at most one consistent typing for a weakly-typed degree 1 term of order 0.*

The next proposition shows that the complexity of evaluating degree 1 weakly-typed terms matches the simply-typed case.

PROPOSITION 27. *Degree 1 weakly-typed terms can be evaluated in PSPACE.*

PROOF. We give a recursive function `Eval` which takes as arguments a subterm of a given term  $\tau$  and a set  $S = \{(c_1, v_1) \dots (c_n, v_n)\}$ , with  $c_i$  positions in binary and  $v_i$  a value in the active domain of the term. By the evaluation of the subterm  $\rho$ , we mean its unique evaluation during a reduction of  $\tau$  (i.e. when free variables in  $\rho$  are replaced by the relations to which they are applied in an innermost reduction). We will arrange that `Eval` returns true exactly when there is some tuple in the evaluation of  $\rho$  whose projection onto each position  $c_i$  is  $v_i$ .  $n$  can be 0, as it is for the top-level call, representing a request to see if the original term  $\tau$  evaluates to a non-empty instance.

- For a subterm of the form  $\sigma_{p=v}\rho$  we return `Eval`( $\rho, S \cup \{(p, v)\}$ ). For  $\sigma_{p_1=p_2}\rho$  we guess a value  $v$  from the active domain of the term and return the conjunction of `Eval`( $\sigma_{p_1=v}\rho, S$ ) and `Eval`( $\sigma_{p_2=v}\rho, S$ ).
- For a subterm  $\rho_1 \times \rho_2$  we calculate the arity of  $\rho_1$  as  $m_1$ . Letting  $S_1$  be the subset of  $S$  consisting of pairs  $(c_i, v_i)$  with  $c_i \leq m_1$ . We return the conjunction of `Eval`( $\rho_1, S_1$ ) and `Eval`( $\rho_2, S - S_1$ ).
- For a subterm  $\rho_1 \setminus \rho_2$  we return `Eval`( $\rho_1, S$ )  $\setminus$  `Eval`( $\rho_2, S$ ).
- For a subterm  $\pi_{[l,u]}\rho$  we return `Eval`( $\rho, \{(c_{1+l}, v_1) \dots (c_{n+l}, v_n)\}$ ).
- For a subterm  $\rho_1 \cup \rho_2$  we return the union of `Eval`( $\rho_1, S$ ) and `Eval`( $\rho_2, S$ ).
- For a subterm  $D_0$ , where  $D_0$  is a data constant, we simply calculate the projection of  $\bigwedge_i \sigma_{c_i=v_i} D_0$  and return true iff this is nonempty.
- Finally, for a subterm consisting of a relational variable  $R$ , we let  $\rho_1$  be the subterm to which  $R$  is applied in  $\tau$ ; we return `Eval`( $\rho_1, S$ ).

The size of the call stack will grow proportionally to the height of the parse tree of the term, and hence is polynomially bounded. Each step of the algorithm can be done in NP. Hence the resulting algorithm will use polynomial space.  $\square$

Furthermore, the algorithm uses a stack of height  $h$  bounded by the nesting-depth of the term, with each stack element requiring space at most  $d \cdot m$ , where  $d$  is the maximal size needed to represent an element of the active domain of the term and  $m$  is the sum of the sizes of positions in the term. In particular, the time used by the algorithm is at most  $h \cdot 2^{d \cdot m}$ .

For a term  $\tau$  of degree  $k = 2$  we proceed by applying  $\beta$ -reduction to get to a term of degree 1.  $d$  and  $m$  do not change during the reduction process, while the height of the resulting degree 1 term is bounded by  $2^{|\tau|}$ .

Hence we have:

PROPOSITION 28. *Degree 2 weakly-typed terms can be evaluated in EXPTIME.*

We believe that this approach can be used to show that the worst case complexity of evaluation for the weakly-typed calculus is the same as that of the strongly-typed one in higher degrees; further exploration of weakly-typed languages is left for future work.

## 7. RELATED WORK

Our work is closely related to a line of research from the 90's in functional databases [HKM93, BF79, OBBT89], aiming at the unification of database query languages with functional programming. Paris Kannelakis and collaborators investigated embeddings of relational query languages into typed  $\lambda$ -calculi. See [HKM93, HK94], and for a good compendium see the ph.d. thesis of Hillebrand [Hil94]). The goal is to code the operational semantics of relational query languages in the standard reduction operations of the host calculus. [HKM93, HK94] 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 [Hil94]). In particular, they show that a standard object-oriented calculus can be embedded into the polymorphic version of the calculus. They also determine the data complexity of evaluation for queries that can be encoded in both the strongly-typed and polymorphic variants of the calculus.

Formally, our language is quite different from those in this prior line, since in our languages we do not *reduce* querying to  $\beta$ -reduction. We simply combine querying and reduction: relational operators 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: the results of [Hil94] 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. This formal distinction notwithstanding, the proof techniques we use to analyze high-degree terms extends the ideas of Hillebrand, Kannelakis, and Mairson. Our results show that the impact of database query constants is localised 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 formalisation of this.

Our work also relates to prior studies of the complexity of nested relational calculus and XML query languages. The Monad Algebra of [TBW92] is presented as a variant of  $\lambda$ -calculus over a type system capturing nested relational structures; one cannot abstract over queries, but one can build up functions from relations via nesting. Koch has shown that these languages are equivalent (modulo coding issues) to the functional XML query language XQuery [Koc06]. [Koc06] has also isolated the complexity of these languages within  $TA[2^{O(n)}, O(n)]$ .

In comparison to our languages these are limited in expressiveness by lacking higher-order abstraction; on the other hand, when compared to our strongly-typed calculus they have some additional succinctness, gained from being able to build up larger and larger intermediate data items of the same type. Our weakly-typed language makes up some of this difference – indeed, this was one of the motivations for introducing it. However, our results show that merely building up high-arity data items is not sufficient, unless you have sufficient expressive power to manipulate them. In future

work we will investigate what exactly needs to be added to the weakly-typed language in order to match the expressiveness of Monad Algebra.

Our complexity bounds in low degree do not have an exact match with prior languages. One intuition for this is: what you can code using database constants and relational operators differs from what you can code either without any constants (as in the ordinary  $\lambda$ -calculus) or what you can code with XML operators.

## 8. CONCLUSIONS

Table 1 summarizes the main complexity results in the paper, excluding those for the weakly-typed languages. We have shown that the evaluation problem has non-elementary complexity, as one might expect from prior results in the  $\lambda$ -calculus. Restrictions on nesting lead to drastic reductions – indeed, a more precise bound in terms of the nesting depth should be derivable from our techniques. Since the upper bounds rely only on an analysis of reduction and the complexity of evaluation of the term algebra over the constants, one can easily accommodate other built-in query transformation and database operations.

| Deg    | Sig.               | General   | Un-nested |
|--------|--------------------|-----------|-----------|
| 1      | RA,RA <sup>+</sup> | PSPACE    | NA        |
|        | RA+ifp             | EXPTIME   | NA        |
| 2      | RA,RA <sup>+</sup> | EXPTIME   | PSPACE    |
|        | RA+ifp             | EXPTIME   | EXPTIME   |
| 2m + 1 | RA,RA <sup>+</sup> | m-EXSPACE | PSPACE    |
|        | RA+ifp             | m-EXSPACE | EXPTIME   |
| 2m     | RA,RA <sup>+</sup> | m-EXPTIME | PSPACE    |
|        | RA+ifp             | m-EXPTIME | EXPTIME   |

Table 1: Complexity of Evaluation

We have initiated a study of weakly-typed terms here; in future work we will consider more powerful operations on indices in weakly-typed terms, with the goal of matching the expressiveness of XML query languages. We will also study the relationship with polymorphic nested relational languages.

Our upper bounds rely on strategies where the interaction between  $\beta$ -reduction and query evaluation is fairly limited. We are not convinced that this is true for practical evaluation strategies, and will be looking at interleaved strategies in our implementation.

**Acknowledgements.** We are very grateful to the anonymous referees of ICDT for helpful comments and corrections. Benedikt is supported in part by EPSRC EP/G004021/1, EPSRC EP/H017690/1 (the Engineering and Physical Sciences Research Council, UK), and FOX: Foundations of XML - Safe Processing of Dynamic Data over the Internet.

## 9. REFERENCES

[BF79] P. Buneman and R. Frankel. FQL: a Functional Query Language. In *SIGMOD*, 1979.

[BPV10] M. Benedikt, G. Puppis, and H. Vu. Positive Higher Order Queries. In *PODS*, 2010.

[DEGV01] E. Dantsin, T. Eiter, G. Gottlob, and A. Voronkov. Complexity and Expressive Power of Logic Programming. *ACM Comput. Surv.*, 33(3):374–425, 2001.

[FGK07] W. Fan, F. Geerts, and X. J. A. Kementsietsidis. Rewriting Regular XPath Queries on XML Views. In *ICDE*, 2007.

[GP03] G. Gottlob and C. Papadimitriou. On the complexity of single-rule datalog queries. *Inf. Comput.*, 183(1):104–122, 2003.

[Hil94] G. Hillebrand. *Finite Model Theory in the Simply Typed Lambda Calculus*. PhD thesis, Brown University, 1994.

[HK94] G. Hillebrand and P. Kanellakis. Functional Database Query Languages as Typed Lambda Calculi of Fixed Order . In *PODS*, 1994.

[HK96] G. Hillebrand and P. Kanellakis. On the expressive power of simply typed and let-polymorphic lambda calculi. In *LICS*, 1996.

[HKM93] G. Hillebrand, P. Kanellakis, and H. Mairson. Database Query Languages Embedded in the Typed Lambda Calculus. In *LICS*, 1993.

[Koc06] C. Koch. On the Complexity of Nonrecursive XQuery and Functional Query Languages on Complex Values. *ACM TODS*, 31(4):1215–1256, 2006.

[LHR<sup>+</sup>10] C. Li, M. Hay, V. Rastogi, G. Miklau, and A. McGregor. Optimizing linear counting queries under differential privacy. In *PODS*, 2010.

[LRU96] A. Levy, A. Rajaraman, and J. Ullman. Answering Queries using Limited External Query Processors . In *PODS*, 1996.

[Mai92] H. Mairson. A simple proof of a theorem of Statman. *Theoretical Computer Science*, 103:387–394, 1992.

[OBBT89] A. Ohori, P. Buneman, and V. Breazu-Tannen. Database programming in Machiavelli—a polymorphic language with static type inference. In *SIGMOD*, 1989.

[RCDS09] J. Robie, D. Chamberlin, J. Dyck, and J. Snelson. XQuery 1.1.: An XML Query Language , 2009.

[Sch01] A. Schubert. The Complexity of  $\beta$ -Reduction in Low Orders. In *TLCA*, 2001.

[TBW92] V. Tannen, P. Buneman, and L. Wong. Naturally Embedded Query Languages. In *ICDT*, 1992.

[Var82] M. Y. Vardi. The Complexity of Relational Query Languages. In *STOC*, 1982.

[VV98] S. Vorobyov and A. Voronkov. Complexity of nonrecursive logic programs with complex values. In *PODS*, 1998.

[Wad71] C. Wadsworth. *Semantics and pragmatics of the lambda-calculus*. PhD thesis, University of Oxford, 1971.