Higher-Order Queries and Applications



Quoc Huy Vu
Wolfson College
University of Oxford

A thesis submitted for the degree of
Doctor of Philosophy
Hilary Term 2012

Dedicated to my wife and my daughter.

Acknowledgments

First of all, I would like to express my deep gratitude and appreciation to Prof. Michael Benedikt, my academic advisor. The weekly meetings with him within more than three years and his precious advice have helped me a lot in writing this thesis.

I also wish to express my best thanks to Dr. Dan Olteanu, Dr. Peter Wood, Prof. Stephan Kreutzer, and Dr. Tim Furche, for reading my transfer report, my confirmation thesis, and this D.Phil. thesis. They have made very constructive comments for further improvements.

I would like to thank Prof. Georg Gottlob, my department advisor, and Dr. Geraint Jones, my college advisor for their useful ideas and support, which have great influence on this work. I also would like to thank Prof. Marta Kwiatkowska and Prof. Luke Ong, for their support and lectures, from which I have gained background knowledge for my thesis and passion for research.

In addition, I am thankful to Gabriele, who has collaborated with me in the first publication during my D.Phil. study, to my office friends, Clemens and Cristian, who often discuss with me about our research.

I would like to thank Prof. Hermione Lee from Wolfson college, Julie, Katie, and Wendy from the Department of Computer Science for their administrative support.

To my parents and all the members of my family, thank you all for your eternal love, understanding, and patience.

Higher-Order Queries and Applications

Quoc Huy Vu
Wolfson College

Doctor of Philosophy, Hilary Term 2012

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. In XML databases, higher-order functions are natural due to the close connection of XML query languages with functional programming.

The thesis investigates higher-order query languages that combine higher-order transformations with ordinary database query languages. We define higher-order query languages based on Relational Algebra, Monad Algebra, and XQuery. The thesis also studies basic problems for these query languages including evaluation, containment, and type inference. We show that even though evaluating these higher-order query languages is non-elementary, there are subclasses that are polynomially reducible to evaluation for ordinary query languages.

Our theoretical analysis is complemented by an implementation of the languages, our Higher-Order Mapping Evaluation System (HOMES). The system integrates querying and query transformation in a single higher-order query language. It allows users to write queries that integrate and combine query transformations. The system is implemented on top of traditional database management systems. The evaluation algorithm is optimized by a combination of subquery caching techniques from relational and XML databases and sharing detection schemes from functional programming.

Contents

1 Introduction **1**

1.1 Motivations 1

1.2 Our contributions 8

1.3 Organization 10

2 Related Work **12**

2.1 Related work 12

2.2 Comparison of our work with the other approaches 14

3 Background **17**

3.1 Background on database queries 17

3.2 Background on complexity 22

4 Higher-Order Queries **24**

4.1 Higher-order languages for relational databases 24

4.1.1 HO: A higher-order language over Relational Algebra 24

4.1.2 Expressiveness and succinctness of HO 33

4.2 Higher-order languages for XML databases and complex values 38

4.2.1 XQH: A higher-order extension of Core XQuery 38

4.2.2 HOCV: A higher-order complex-valued query language 43

4.2.3 Expressiveness and succinctness of HOCV and XQH 46

5 Evaluation of Higher-Order Terms **50**

5.1 The evaluation problem 51

5.2 Complexity of evaluating HO 52

5.2.1 Complexity of evaluating lower degree terms 52

5.2.2 Complexity of evaluating arbitrary degree terms 58

5.2.3 Reducing the complexity of evaluation 67

5.3 Complexity of evaluating HOCV and XQH 71

5.4 Conclusions 81

6 Containment between Higher-Order Queries 83
6.1 The containment problem 84
6.2 Containment between HO terms of order 1 88
6.2.1 Containment between lower degree terms 88
6.2.2 Containment between higher degree terms 94
6.3 Containment between normalized HO terms of order 2 94
6.3.1 Complexity of higher-order containment 94
6.3.2 Tractable cases 108
6.3.3 Higher-order containment in full Relational Algebra base . . . 111
6.3.4 Higher-order containment in CQ base 112
6.4 Containment between HOCVP terms of order 1 119
6.5 Containment between normalized HOCVP terms of order 2 124
6.6 Conclusions 127

7 Typing Problems 129
7.1 Introduction 129
7.2 Typeability for higher-order languages 129
7.3 Weakly-typed HO terms 132
7.4 Conclusions 135

8 Higher-Order Query Language Implementation 137
8.1 Motivation and introduction on graph reduction 137
8.2 System architecture 140
8.3 Optimization 142
8.4 The differences between relational implementation and XML implementation 148
8.5 Empirical evaluation 150
8.6 Prototype details 156
8.7 Conclusions 156

9 Conclusions and Future Work 159
9.1 Conclusions 159
9.2 Future work 160

References 163

List of Figures

4.1 Syntax of the higher-order query language HO 25
4.2 The construction tree of τ_0. 33
4.3 Syntax of the higher-order query language XQH 38

8.1 Graph reduction with sharing. 138
8.2 Graph reduction with sharing and copying. 139
8.3 The construction tree and the graph reduction stage of τ. 141
8.4 A screen-shot of the prototype. 157
8.5 A screen-shot of graph reduction generated in the prototype. 158

List of Tables

1.1 Relation between chapters and material. 10

5.1 Complexity of evaluation for HO. 81
5.2 Complexity of evaluation for HOCV and XQH 82

6.1 Complexity of containment. 84

8.1 Empirical evaluation results for the relational data case. 151
8.2 Empirical evaluation results for HO queries over the table *route*. . . . 153

Chapter 1

Introduction

1.1 Motivations

Higher-order functions play a fundamental role in computer science, and most functional programming languages feature them. One of our motivations is to find a uniform approach for integrating query transformation into query languages. In database systems higher-order functions have appeared in isolation at several points: query-transformation plays a role in numerous aspects of databases, including processing queries over views, query relaxation, query specification and access control.

One of the approaches to processing queries over views is via *query rewriting*. Levy et al. [LMSS95] initiated the approach of query rewriting, where a query over a set of base tables is rewritten in terms of views before being evaluated. There are a number of research results on query rewriting where the queries to be rewritten and the views range from conjunctive queries (CQs), Datalog queries, to First-Order logic (FO) [DG97, GM99, SV05, NSV07, Hal00]. Recently, query rewriting for XML data has received a lot of research interest, e.g., [ODPC06, FGK07]. We can consider processing queries over views as a rewriting higher-order operator that maps a fixed-length sequence of view definitions and a user query to a rewritten query. An example of how this rewriting higher-order operator is integrated in our language will be given in Example 3.

Query relaxation reduces the conditions in a user query into a less restricted form in order to have more appropriate answers [KLTV06, AYCS02]. Koudas et al. [KLTV06] use query relaxation to adjust a particular query to obtain a non-empty output; whereas, Amer-Yahia et al. [AYCS02] adjust the queries to return more answers. Again, query relaxation can be seen as a query-transformation.

Another topic of recent interest is *query specification and access control* [LRU96, VP00, CDO09]. Due to intellectual property reasons, load-control or privacy reasons,

data sources are often restricted so that only a family of parameterized queries is permitted to be answered. Query specification describes the set of queries that any user can run against a particular data source. A number of description languages, based on Datalog, have been proposed to describe all possible queries that the sources are allowed to accept [LRU96, VP00]. Recently, Cautis and collaborators [CDO09, CDOV11] have distinguished several versions of query specification and extended this to XML data sources. Abstractly, these specifications can be seen as functions taking as input a query and returning true or false, corresponding to whether the query is permitted or not.

We would like a language that can capture both ordinary queries and a variety of higher-order functions, in a compositional way. While programming languages provide many facilities to develop and test software in modules, query languages provide much less support for modularity. The language which integrates higher-order functions allows users to write more complex queries and to interact more with programming languages. This language provides the modularity and allows users to reuse some queries in more complicated ones.

By integrating query transformations with ordinary queries in a query language, new kind of queries can be designed and better evaluation techniques can be implemented. Ordinary querying environments provide no support for interactively evaluating queries, progressively evaluating each as more of their constituent components are defined. Our language will deal with the problem of partially evaluating queries – evaluating them maximally, given that parts of the queries are not yet defined. One can use this language to design complex queries, which contain incomplete elements and can be refined interactively. The incomplete elements are considered as templates for additional parameters. Even though the queries are incomplete, an evaluation system can partially evaluate the complete parts of the queries. When more parameters are provided to the queries, the evaluation system will automatically evaluate the new complete parts. A simple scenario of this is given in Example 1.

A second motivation for higher-order querying comes from creating better integration of querying and functional programming. While relational query languages, such as SQL, have not generally had a close connection to functional programming languages, 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, because the main XML

query language, XQuery [RCDSa, HPVD04], is functional, with close connections to functional query languages.

There is already a known connection between Core XQuery and higher-order functions. Koch [Koc06] has shown a correspondence between the complex-valued query language *Monad Algebra* and Core XQuery; Monad Algebra includes a restricted λ-calculus in it, allowing the definition of higher-order terms. The current version of these languages only allows users to define abstraction over data, e.g., relations, nodes, node sequences, and queries. As such, it is useful to consider support for higher-order queries, such as transformations of queries, and transformations of transformations of queries, within complex-valued query languages and XQuery. Actually, the XQuery 1.1 Recommendation [RCDSa] contains a proposal for higher-order support. However, there has been little formal work on such extensions, and no study of complexity issues for basic algorithms.

In summary, the importance of query transformations makes it natural for us to consider a query language for querying queries. We refer to this as a *higher-order query language*. As we have seen in our motivations above, higher-order transformations are ubiquitous in computer science, but the combination of database queries and higher-order functions has not been throughly studied in its own right. We look for languages with two important properties. The first is that the transformations defined in our languages, as in functional programming, are *generic* – the output of a term when the higher-order variables are bound to queries depends only on the semantics of the queries. This is in contrast to query transformation and specification languages which allow direct access to the syntax of the queries [NVdBVGV99, VP00]. Secondly, we search for languages where *static analysis and optimization are possible*, extending techniques from the case of standard selection project join queries in the relational case. This is again in contrast to prior languages for querying queries (e.g., [NVdBVGV99]), which are relationally complete, and hence cannot admit static guarantees even of satisfiability.

In this thesis, a straightforward framework for combining ordinary queries with higher-order functional languages is defined: it is exactly *the simply-typed λ-calculus* with data instances, certain fixed first and second order operators as "constants" (that is, as built-in functions). The terms in these higher-order query languages feature both variables ranging over queries and variables ranging over data instances. We will consider the use of our higher-order languages through a number of examples below, which show a few properties of the languages.

We start with examples of the higher-order language for relational data. Recall that one motivation for higher-order languages is to provide modularity and convenience in designing and programming systems. We consider an example where we need an interface to control query access to a relational instance.

Example 1. *Given a source R_1 containing two columns title and year, and a query Q, accesses to R_1 via Q are transformed for security reasons, returning the result of Q on only tuples with year > 1900 in R_1 and returning only the column title, in the output of Q. This could be implemented via the following higher-order query.*

$\tau_0 := \lambda R_1.\lambda Q.\ SELECT\ title\ FROM$

$$Q(SELECT\ *\ FROM\ R_1\ WHERE\ year > 1900)$$

The query above is written even though we have not known all the information of R_1 and Q. When we know that the information source for R_1 is a table Book, we can apply τ_0 on Book to get the following query:

$\tau_0 := \lambda Q.\ SELECT\ title\ FROM$

$$Q(SELECT\ *\ FROM\ Book\ WHERE\ year > 1900)$$

*Even though Q is not defined, an evaluation system can evaluate the subquery SELECT * FROM Book WHERE year > 1900 and store the output to a table Book1. When the information about Q is given, the whole query $\lambda Q.\ SELECT\ title\ FROM\ Q(Book1)$ will be evaluated.*

Example 2. *In Example 1, the subterm SELECT * FROM R_1 WHERE year > 1900 filters the input data; for modularity we may wish to develop the query without a particular filter in mind. We can thus create a more "generic" higher-order filtering query, with a query variable Fil representing a filter:*

$\tau_0' := \lambda Fil.\lambda R_1.\lambda Q.\ SELECT\ title\ FROM\ Q(Fil(R_1))$

Later when we are ready to commit to using fil_0, we can reclaim τ_0 as $\tau_0'(fil_0)$ with

$fil_0 := \lambda R_2.(SELECT\ *\ FROM\ R_2\ WHERE\ year > 1900)$.

The above is in an SQL syntax that we use for readability only in the motivation section. In the thesis we will use Relational Algebra-based syntax, and the terms τ_0, τ_0', fil_0 in the example would be written as follows.

$\tau_0 := \lambda R_1.\lambda Q.\pi_{\{title\}}(Q(\sigma_{year>1900}R_1))$

$\tau_0' := \lambda Fil.\lambda R_1.\lambda Q.\pi_{\{title\}}\,(Q(Fil(R_1)))$

$fil_0 := \lambda R_2.\sigma_{year>1900}R_2$

In the remaining relational examples, we will use both the Relational Algebra syntax and the SQL syntax.

The higher-order languages can integrate the results of higher-order operators with other operators. Note that our higher-order languages cannot express these higher-order operators, but they simply integrate higher-order operators as "black boxes". The integration nevertheless still benefits the query design and evaluation process. As mentioned above, a very common example of higher-order transformations in database management is query rewriting. For example, given a query Q and a set of views $V_1, \ldots, V_n$ both over $D_1, \ldots, D_m$, we may be interested in generating the maximally contained rewriting of Q over $V_1, \ldots, V_n$. There are many algorithms for obtaining these rewritings (e.g., Bucket [LRO96b], MiniCon [PH01]). They can be encapsulated as an operator, called RW, that takes a query and a set of views as its input, and returns a rewriting of the query. In a higher-order querying system, users can freely make use of RW in building more complex queries.

Example 3. *A user could write a higher-order term that takes queries Q, Fil, V_1, V_2 and first rewrites Q with respect to V_1 and V_2 and then post-filters the result using Fil.*

$$\tau_1 := \lambda V_1.\lambda V_2.\lambda Q.\lambda Fil.\tau_0'(Fil)(\mathsf{RW}(Q, V_1, V_2))$$

where τ_0' is the "post-filtering transform" of Example 2. Later they can instantiate the views, forming τ_2 by applying τ_1 to

$$V_1 = \lambda R_2.\ SELECT\ *\ FROM\ R_2\ WHERE\ R_2.print_year > 1900$$

$$V_2 = \lambda R_2.\ SELECT\ *\ FROM\ R_2\ WHERE\ R_2.sale_year < 2000$$

Still later they can instantiate Q, forming τ_3 by applying τ_2 to:

$$Q = \lambda R_2.\ SELECT\ *\ FROM\ R_2\ WHERE\ R_2.print_year = R_2.sale_year$$

and finally they can instantiate Fil by forming $\tau_3(fil_0)$, where fil_0 is the filter from Example 2.

A strength of our languages is that they allow users to write queries very succinctly. In the following example, we write a "small size" term that checks the existence of a path of length doubly exponential in the size of the term.

Example 4. *A relation with integer attributes* $(from, to)$ *can code a graph. Let* τ_p^2 *be an ordinary conjunctive query returning paths of length* 2 *in such a relation: such a query is easily written as a self-join.*

$$\tau_p^2 = \lambda R_0.\pi_{\{from,to\}}(\rho_{to/transit}(R_0) \bowtie \rho_{from/transit}(R_0))$$

In τ_p^2, R_0 *represents a relation that* τ_p^2 *requires as its input.*

Consider the term.

$$\tau_p^{16} = (\lambda Q_1.\lambda R_1.(Q_1(Q_1(R_1))))((\lambda Q.\lambda R.(Q(Q(R))))\tau_p^2)$$

The subterm $((\lambda Q.\lambda R.(Q(Q(R))))\tau_p^2)$ *of the term* τ_p^{16} *applies two nestings of* τ_p^2 *on an input* R *because* Q *is substituted by* τ_p^2. *Thus, this subterm returns all the pairs of nodes having a path of length 4 between them. Similarly,* $(\lambda Q_1.\lambda R_1.(Q_1(Q_1(R_1))))$ *applies two nestings of the subterm* $((\lambda Q.\lambda R.(Q(Q(R))))\tau_p^2)$ *on an input* R_1 *because* Q_1 *is substituted by* $((\lambda Q.\lambda R.(Q(Q(R))))\tau_p^2)$. *Therefore,* τ_p^{16} *takes as input a graph and returns a graph containing all the pairs of nodes having a path of length 16 between them.*

In general, using query variables, we can write queries of length n *checking for paths of length doubly exponential in* n.

We now give examples of higher-order queries for XML data. Notice that our syntax will be different from the examples above to be more consistent with XQuery rather than Relational Algebra. Specifically, the notation $\lambda X_1 \ldots \lambda X_n$ with $X_1, \ldots, X_n$ variable names is written as $[\$X_1, \ldots, \$X_n]$.

Increase of modularity is a traditional motivating scenario for higher-order functions in XQuery. The following example is taken from a whitepaper by an XQuery working group member on adding higher-order query support to XQuery [Sne10]. We want to build a "generic sorting query" which sorts a sequence by the sort key defined by a user provided function. In our higher-order version of XQuery, we proceed as follows.

Example 5. *We define the generic function.*

$$\mathsf{sort} := [\$seq, \$key]\{for\ \$a\ in\ \$seq\ order\ by\ \$key(\$a)\ return\ \$a\}$$

[$seq, $key] declares that this is a function of two arguments, the first being a node sequence and the second an input representing an "arbitrary" function of the appropriate type.

Then we apply sort *to a particular sequence and function:*

$$query_0 := \mathsf{sort}(doc(\text{“}books.xml\text{”})/book)([\$x]\ \$x/title)$$

Above $doc(\text{“}books.xml\text{”})/book$ *returns a sequence of books having title, author, and year children, while* $[\$x]\ \$x/title$ *is a function that takes as argument a book* $\$x$ *and returns the title. The output of this term is a sequence of books ordered by their titles.*

The higher-order query above is written in XQuery 3.0 syntax [RCDSb] as follows.

```
declare function local:sort($seq as item()*, $key as function(item()*) as item()*)
                            as item()*
{
     for $a in $seq order by $key($a) return $a
};
let $f := function($x) {$x/title}
return
     local:sort(doc('books.xml')/book, $f)
```

Transformations of queries play an important role in XML access control and data integration [FGK07]. In the next example, we consider the situation where we need an interface to control the access of a query over an XML sequence.

Example 6. *Given a sequence* $\$seq$ *and a query* $\$Q$*, accesses to* $\$seq$ *via* $\$Q$ *are transformed for security reasons, returning the result of* $\$Q$ *on* $\$seq$ *only after it is filtered. However, we may wish to develop the query without a particular filter in mind. This could be implemented via the following expression in our higher-order XQuery language:*

$$query_1 := [\$fil, \$Q, \$seq]\ \$Q(\$fil(\$seq))$$

The notation $[\$fil, \$Q, \$seq]$ *is a declaration that there are three arguments – equivalently, it is a sequence of three* λ*-abstractions.*

Later, we can partially evaluate using the following query as the filter:

$$fil_0 := [\$x]\ \{for\ \$y\ in\ \$x\ where\ \$y/year > 1990\ return\ \$y\}$$

creating the higher-order XQuery $query_2 := query_1(fil_0)$ *that returns the result of* $\$Q$ *on only a selection of* $\$seq$*. Note that* $\$fil$ *occurs first in the list* $[\$fil, \$Q, \$seq]$*, so* $\$fil$ *will be substituted by* fil_0*. Similarly,* $\$x$ *in* fil_0 *is substituted by* $\$seq$ *when we consider the subterm* $\$fil(\$seq)$*. The reduced form of* $query_2$ *is as follows.*

$$\begin{aligned} query_2 \quad &= [\$Q, \$seq]\ \$Q(\\ &\quad \{for\ \$b\ in\ \$seq\ where\ \$b/year > 1990\ return\ \$b\}) \end{aligned}$$

Although there is no formal semantics for XQuery 3.0, in several implementations we have looked at, such as BaseX [GHS07], the mechanism for creating functions that can be passed as arguments into higher-order queries is via let *expressions. One could model the example above in such an implementation as follows:*

```
declare function local:query2($Q as function(item()*) as item()*, $seq as item()*)
                              as item()*
{
     let query1 := function($fil, $Q, $x)
          { $Q($fil($x)) }

     let $fil0 := function($y)
          { for $b in $y  where $b/year > 1990 return $b }

     return
          $query1($fil0, $Q, $seq)
};
```

We will explain the syntax and semantics of the higher-order languages for relational, XML, and complex-valued data in detail later on in the thesis. After defining the higher-order query languages, the thesis studies fundamental problems: query evaluation, containment and equivalence between queries, and query typeability. We will show that the worst-case complexity of evaluation of the language is non-elementary, and so no evaluation strategy can be efficient on every query. We will look at subclasses where better worst-case bounds are obtainable. On the pragmatic side, we discuss our implementation that uses specialized techniques which combine β-reduction from functional programming with materialization from database processing to increase efficiency.

1.2 Our contributions

Higher-Order languages for relational databases, complex values, and XML databases.
We introduce a natural way to add higher-order functionality to query languages, by adding database query operators and data instances to the λ-calculus as constants. This framework, which we refer to as *λ-embedded query languages*, embeds data instances and database queries within the simply-typed λ-calculus. Our language allows one to succinctly define ordinary database queries and, in addition, *second-order query functionals*, which allow the transformation of database queries in a generic (i.e., syntax-independent) way. We define higher-order queries languages as extensions of Relational Algebra, Monad Algebra, and Core XQuery, which is a fragment

of XQuery defined by Koch in [Koc06]. We also study the relationship between these higher-order query languages and the ordinary query languages.

The evaluation problem.
We take an in-depth look at the most basic issue for such languages: the evaluation problem. For all the higher-order languages for relational databases, complex values, and XML databases, we give a fairly complete picture of the complexity of evaluation for these λ-embedded query languages.

For the higher-order language over relational databases, we look 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. We give tight bounds on both the combined complexity and the query complexity of evaluation in all these settings.

Extending Koch's correspondence in [Koc06], we give a polynomial reduction between the evaluation problems for the complex-valued higher-order language and for the XML higher-order language. Thus, the complexity of evaluating the higher-order extension for XML databases is shown by studying the evaluation problem for the higher-order query language for complex values.

For all the higher-order query languages, we isolate cases of the languages that require lower complexity.

The containment problem.
We study two different versions of the containment problem for higher-order queries. The first one is containment between two terms that evaluate to ordinary queries, e.g., Relational Algebra expressions and Monad Algebra expressions. The second one is between two terms that evaluate to query transformations. In both versions, the higher-order terms can contain variables of higher degree. We only consider the containment problem for the positive fragments, because the containment between full Relational Algebra expressions is already undecidable.

The first version of the problem subsumes traditional query containment. We give the complexity of checking the containment between arbitrary higher-order terms.

For the second version, we have to define what equivalence and containment mean for query functionals. Query functionals are said to be equivalent if the output queries are equivalent, for each possible input query, and similarly for containment. This problem depends on the class of queries considered as inputs, which is an additional parameter to the containment problem. We show that containment and equivalence are decidable when query variables are restricted to positive queries (e.g., in the

relational case, to positive Relational Algebra) and we identify the precise complexity of the problem. We also identify classes of functionals where containment is tractable.

The type inference problem.
We study a generalization of the typeability problem for our higher-order query languages. In the general typeability problem, where one can assign types for any part of a term, we check if there exists an assignment for the other constants and variables to make the term valid. We also consider a special case of this problem when the type of all constants are given, which makes the problem tractable.

Higher-order query language implementation.
Particular higher-order functions, such as query-rewriting transformations, or even flexible rule-based query-rewriting frameworks, have been implemented stand-alone for decades. But their implementation is not part of a system that integrates higher-order transformations with ordinary data transformations, as in functional programs. Functional databases allow the definition of higher-order terms, but do not support query transformation. We develop Higher-Order Mapping Evaluation System (HOMES), an evaluation system for the higher-order queries introduced above.

Origin of the presented material.
The thesis contains many parts of published and submitted material of which I am a co-author. Table 1.1 summarizes the relation between chapters and the material.

Chapter	Material	Co-author
Chapter 4	[VB11a], [BV12]	Michael Benedikt
Chapter 5		
Part of Chapter 7		
Part of Chapter 6	[BPV10]	Michael Benedikt, Gabrielle Puppis
Part of Chapter 8	[VB11b]	Michael Benedikt

Table 1.1: Relation between chapters and material.

1.3 Organization

The remainder of the thesis is organized as follows.

Chapter 2 reviews the work that is related to the thesis's topic.

Chapter 3 gives some background on the ordinary query languages for the higher-order queries introduced in Chapter 4, and complexity classes used in the thesis.

Chapter 4 presents our higher-order languages over relational databases, XML databases, and complex values. It also gives a polynomial reduction between higher-order queries over complex values and over XML databases.

Chapter 5 studies the complexity of evaluating higher-order terms.

Chapter 6 considers the containment problem for higher-order terms.

Chapter 7 studies the type inference problem for higher-order terms. This chapter shows the complexity of checking the typeability of higher-order terms.

Chapter 8 describes the implementation and demonstration of a system evaluating higher-order terms.

Chapter 9 concludes the thesis and proposes future directions.

Chapter 2

Related Work

2.1 Related work

Current integration of higher-order queries to other database query languages.
There is a line of research from the 90's in functional databases [HKM93], aiming at the unification of database query languages with functional programming. Kannelakis and collaborators investigated embeddings of relational query languages into typed λ-calculi [HKM93, HK94, HK96]. (For a good compendium, see the Ph.D. thesis of Hillebrand [Hil94]). The goal is to encode the operational semantics of relational query languages in the standard reduction operations of the host calculus. Hillebrand et al. [HKM93] give polynomial time encodings of standard languages, including query languages with recursion mechanisms, within variants of the λ-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.

Languages such as Machiavelli [OBBT89] and Kleisli [Won00] 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.

Recently, Cooper [Coo09] has defined a higher-order language that integrates λ-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 [Ulr11] has described an implementation that uses the FERRY framework [GMRS09] to translate a subset of the LINKS programming language [CLWY07], 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.

Tannen et al. [TBW92] present Monad Algebra as a λ-calculus over a type system capturing nested relational structures. Rather than embed into a general-purpose calculus, they allow functions to be built up via a collection of nested relational operators. Koch [Koc06] 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 [PVG92] implies that the expressive power of such a language on relational data is no more than that of relational calculus.

Meta-data and higher-order querying.
Several researchers have looked at the issue of uniformly handling data and meta-data within a query language – particularly see [LSS93, NVdBVGV99, CKW93, Ros92, Ros94]. 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 [NVdBVGV99], which gives a language that can manipulate tables containing both queries and data. The language HiLog of Chen et. al. [CKW93] deals with higher-order logic programming. Wood [Woo93] has extended Datalog evaluation techniques to produce an evaluation mechanism for DataHiLog, which is a restriction of HiLog.

Evaluation and containment problems.
Evaluation of relational queries is a traditional problem, which can be found in many documents, e.g., [AHV95]. Koch [Koc06] has shown the NEXPTIME lower bound and EXPSPACE upper bound of the evaluation problem for a fragment of XQuery and for Monad Algebra. In 1979, Statman [Sta79] proved that typed λ-calculus is non-elementary. A simple proof of Statman's result is later given by Mairson [Mai92]. 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 [HK96].

Query equivalence and containment have been studied extensively for many relational query classes: e.g., conjunctive queries and union of conjunctive queries, starting with [CM77]. Recently, Benedikt and Gottlob [BG10] have shown 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 [LS97] investigate containment and equivalence between queries on complex objects. Later work of Dong et al. [DHT04] studies the containment problem for nested XML queries. Björklund et al. [BMS11] have shown a full picture of the complexity of the containment problem for conjunctive queries over trees.

Typing problems.
The typing problems for higher-order query languages have been addressed before. In the relational setting, a lot of works have defined higher-order query languages [OB88, OBBT89, Wan87a, Mai89], and studied a variety of typing issues involved. They have not looked at the XML setting (indeed, much of their work was pre-XML); in addition, the set of relational operators they allow is limited. As shown in later work, the typing problem for relational query languages is very sensitive to the exact set of relational operators. Van den Bussche and Waller [VdBW02a] study the typing problem for relational languages without higher-order support. Vansummeren gives more precise bounds for this problem [Van05b], and extends it to the setting of XQuery [Van05a]. Van den Bussche et al. [VdBVGV05] look at varieties of the typing problem for Nested Relational queries and XML.

2.2 Comparison of our work with the other approaches

Compared to the work of Kannelakis and collaborators [HKM93, HK94, HK96], our languages are designed with a different approach. In our languages we do not encoding standard languages by only variants of the λ-calculus. We simply combine queries and λ-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 λ-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.

The languages Machiavelli [OBBT89], Kleisli [Won00], and the works of Cooper [Coo09], Ulrich [Ulr11] are much more similar to ours, but there are some technical differences of their frameworks from ours. The lower-order terms of their languages do not match to ordinary queries because they do not define sets of constants as ordinary queries as we do. Additionally, none of these works deal with higher-order functions for XQuery. More importantly, they do not study the complexity of the related problems.

The Monad Algebra, defined in [TBW92, Koc06], is analogous to our languages but there are two main differences. First, 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 λ-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 [Avi03]). Koch [Koc06] 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 [BK09] 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.

Our languages are incomparable with meta-data querying languages [LSS93, Ros92, Ros94, CKW93, Woo93, NVdBVGV99], 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.

The evaluation problems we deal in the thesis are different from prior works because as analyzed above, our higher-order query languages allow abstraction over

higher-order queries including ordinary queries, and functions from queries to queries. For example, in the Monad Algebra of [TBW92] all variables range over database instances – query variables and λ-abstraction over queries are not supported.

Even though equivalence and containment has been studied for relational queries, to the author's knowledge, the containment problem between higher-order queries has not been studied before. Additionally, our higher-order containment problems have no natural analog in the existing functional query literature. In the thesis, we use the definition of containment between complex objects of Levy and Suciu [LS97]. However, their work is not for higher-order terms, and their complexity results are different from ours.

The works about typing reviewed above [OB88, OBBT89, VdBW02a, Van05b, Van05a, VdBVGV05] look at more varieties of the typing problem than we consider in the thesis. For example, they consider the problem of obtaining the "most general" type, rather than merely determining whether there is some typing. They also consider the "well-definedness problem" – a more lenient and less syntactic condition than typeability, which allows some subterms to be untypeable. Our work looks at the most basic typing problem, but for languages combining higher-order functions with structured datatypes.

Chapter 3

Background

3.1 Background on database queries

This section reviews basic definitions in database theory mentioned in the remainder of the thesis. The fragment of XQuery considered in the thesis will be deferred until its use in Subsection 4.2.3 in Chapter 4. First of all, we consider the definition of the relational data model.

We fix a countably finite set of attributes *att*, which contains a total ordering $\leq_{att}$. A countably finite set **dom**, which consists of *constants*, is also fixed. Additionally, *relname* is a countably finite set of *relation names*. Note that *att*, **dom**, and *relname* are disjoint sets.

We define a function *sort* to associate each relation name R with a finite set of attributes $sort(R)$. A *relation schema* is a relation name; whereas, a *database schema* $\mathbf{R}$ is a nonempty finite set of relation schemas.

A *tuple* is defined as a mapping from a finite subset of *att* to **dom**. In addition to the named perspective where attributes have names, there is unnamed perspective where names of attributes are ignored. Because of $\leq_{att}$, there is a natural transformation between the unnamed perspective and the named perspective.

Given a relation R having the attribute set U, its *relation instances* are finite sets of tuples on U. A *database instance* of a database schema $\mathbf{R}$ is a mapping $\mathbf{I}$ which assigns a relation instance to each relation R in $\mathbf{R}$.

In order to query a database, we use different kinds of query languages. Since the thesis works with query languages, it is natural to quickly review basic query languages. Two common query languages in database theory, which are related to the work of this thesis, are *rule-based conjunctive queries* and *Relational Algebra*.

Rule-based conjunctive queries.

The language of rule-based conjunctive queries stems from mathematical logic and

is non-procedural. In the definition, **var** will be used to denote an infinite set of variables that range over **dom**. We recall the definition of these queries from a book written by Abiteboul et al. [AHV95].

Definition 1. *[AHV95] Let* $\mathbf{R}$ *be a database schema. A rule-based conjunctive query over* $\mathbf{R}$ *is an expression of the form*

$$ans(u) \leftarrow R_1(u_1), \ldots, R_n(u_n)$$

where $n \geq 0$*,* $R_1, \ldots, R_n$ *are relation names in* $\mathbf{R}$*; ans is a relation name not in* $\mathbf{R}$*; and* $u, u_1, \ldots, u_n$ *are free tuples (i.e., may use either variables or constants). Recall that if* $v = \langle x_1, \ldots, x_m \rangle$*, then* $R(v)$ *is a shorthand for* $R(x_1, \ldots, x_m)$*. In addition, the tuples* $u, u_1, \ldots, u_n$ *must have the appropriate arities (i.e.,* u *must have arity of ans, and* u_i *must have the arity of* R_i *for each* $i \in [1, n]$*). Finally, each variable occurring in* u *must also occur at least once in* $u_1, \ldots, u_n$*. The set of variables occurring in* q *is denoted by* $var(q)$*.*

A rule-based conjunctive query, which is often called a *rule*, is evaluated as follows. A *valuation* of the subset V of **var** is a total function ν from $V \cup \mathbf{dom}$ to **dom**, which is the identity on **dom**. Using the natural method, we extend the function ν to map free tuples to tuples.

$$q(\mathbf{I}) = \{\nu(u) \mid \nu \text{ is a valuation over } var(q) \text{ and } \nu(u_i) \in \mathbf{I}(R_i) \text{ for each } i \in [1, n]\}$$

where $\mathbf{I}$ is an instance of $\mathbf{R}$. The set of all constants which occur in a database instance $\mathbf{I}$ (resp., a query q) is called the active domain of $\mathbf{I}$ (resp., q), denoted $adom(\mathbf{I})$ (resp., $adom(q)$). Since $R_1, \ldots, R_n$ are given by $\mathbf{I}$, they are called *extensional* relations. Whereas, *ans* is called an *intensional* relation. A set of rule-based conjunctive queries with a common intensional relation is used when the Union operator is added.

Note that a rule-based conjunctive query can be written in *relational calculus* as follows. We consider a conjunctive query of the following form:

$$ans(u) \leftarrow R_1(u_1, v_1), \ldots, R_n(u_n, v_n)$$

such that for all $i \in [1, n]$, each variable occurring in u_i must also occur in u, and all variables occurring in v_i must not occur in u. The query above is rewritten to an equivalent query in relational calculus as below.

$$\{u \mid \exists v_1, \ldots, v_n \;\; R_1(u_1, v_1) \wedge \ldots \wedge R_n(u_n, v_n)\}$$

The formulas in relational calculus include other operations, e.g., negation. We omit the details of relational calculus, a formal definition of which can be found in [Cod72, AHV95].

Relational Algebra.

The second query language comes from an algebraic perspective. The SPJR Algebra contains four primitive operations: selection, projection, (natural) join, and renaming [ASU79b]. The unary selection operator σ_c selects a subset of the tuples from a given relation according to the condition c. The unary projection operator π_A projects an input relation into the subset A of its attributes. The binary join operator $\bowtie$ 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. The output of the join operator also removes all the duplicated attributes, which exist due to the cartesian product of the two input relations. The unary renaming operator $\rho_{a/b}$ renames the attribute a by b in a given input relation. The SPJR Algebra is as expressive as conjunctive queries.

When adding the binary union operator $\cup$, which returns the union of two input relations, to the SPJR Algebra, we have SPJRU Algebra.

When SPJRU is supplemented with the *difference* operator $\setminus$, it is often referred to as *full Relational Algebra* (RA).

Conjunctive query containment.

Q_1 is contained in Q_2, commonly denoted $Q_1 \subseteq Q_2$, iff for any database instance $\mathbf{I}$ the output of $Q_1(\mathbf{I})$ is a subset of $Q_2(\mathbf{I})$.

Q_1 and Q_2 are equivalent, denoted $Q_1 \equiv Q_2$, iff $Q_1 \subseteq Q_2$ and $Q_2 \subseteq Q_1$.

The problem of checking the containment between two conjunctive queries is NP-complete [ASU79b, CM77]. That problem becomes more difficult when inequalities are included in the two conjunctive queries. Klug [Klu88] shows that the containment problem between two CQs with inequalities is in Π_2^{P}. It was proved to be Π_2^{P}-complete by Van der Meyden [VdM92]. When the union operator is included in the SPJR Algebra, Sagiv and Yannakakis [SY80] show that the containment problem is also Π_2^{P}-complete. Yannakakis [Yan81] has considered *acyclic queries*, where the containment problem between two CQs is the tractable.

Conjunctive query containment under dependencies.

Below are the definitions of two main kinds of dependencies given in [ASU79b, JK84].

Definition 2. *[ASU79b, JK84] A* functional dependency *(FD) is a statement $X \to Y$, where X and Y are sets of attributes. A relation R satisfies this functional*

dependency (FD) if and only if for all tuples μ and ν in R the following condition holds: If $\mu(A) = \nu(A)$ for all A in X, then $\mu(B) = \nu(B)$ for all B in Y. That is, if two rows of R agree on the columns for X, then they must agree on the columns for Y.

An inclusion dependency *(ID) is a formal statement of the form $R[X] \subseteq S[Y]$, where R and S are relation names, X is an ordered list of attributes of R, Y is an ordered list of attributes of S of the same length as X. A database satisfies the inclusion dependency $R[J_1, \ldots, J_j] \subseteq S[K_1, \ldots, K_j]$ if for every subtuple $\langle a_1, \ldots, a_j \rangle$ that occurs in columns $J_1, \ldots, J_j$ of some tuple in relation R, there is a tuple of relation S that contains $\langle a_1, \ldots, a_j \rangle$ in column $K_1, \ldots, K_j$.*

The complexity of the containment problem between two conjunctive queries when FDs are added is first considered by Chandra and Merlin [CM77]. They have shown that when there are only FDs, the complexity of the containment problem is NP-complete.

When there are IDs (even when there are no FDs), Casanova et al. [CFP82] show that the conjunctive query containment problem is PSPACE-complete.

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 $R(x_1 \ldots x_n)$ 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 ϕ is a conjunction of atoms over S, and A is an atom over S. A is the *head* of the rule and ϕ is the *body* of the rule. We will often identify ϕ with the set of atomic formulas in it, writing $A_1(\vec{x}_1), \ldots, A_n(\vec{x}_n)$ instead of $A_1(\vec{x}_1) \wedge \ldots \wedge A_n(\vec{x}_n)$. A variable that occurs in the head of a 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 stratified 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 $\mathsf{Rk}(P)$, as follows: the rank of an input predicate is 0, the rank of an intensional predicate P is

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

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 $\mathsf{Rk}(P)$. For P an input predicate, $P(D)$ is the interpretation of P in D. For P an intensional predicate with $\mathsf{Rk}(P) = k + 1$ 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:

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

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.

Vardi [Var82] has implicitly shown that the query complexity and the combined complexity of evaluation for NRDL queries and for NRDL$^\neg$ queries are PSPACE-complete. Benedikt and Gottlob [BG10] show that the complexity of containment between two NRDL queries is co-NEXPTIME-complete.

Note that nonrecursive Datalog with stratified negation can be translated in polynomial time (over models of size two) into First-Order logic or Relational Algebra [Avi03, VV97]. Nonrecursive Datalog translates into positive existential First-Order

logic in (provably worst case) exponential time; this in turn translates into union of Conjunctive Queries, again in exponential time. The number of the conjunctive queries in the union is doubly exponential in the size of the nonrecursive Datalog.

3.2 Background on complexity

We assume familiarity with Turing machines and standard complexity classes, e.g., NP, PSPACE, EXPTIME, EXPSPACE. Their definitions can be found in, e.g., [Joh90]. In our complexity results, we use $\mathtt{exp}_m^n$ to represent 2 to the *m-hyperexponential* of n, i.e., $2^{2^{\cdot^{\cdot^{2^n}}}}$ with a tower of m 2's. Assuming that n is the input size of problems, m-EXPTIME (resp., m-EXPSPACE) denotes the complexity class of problems which are solvable by deterministic Turing machines in $\mathtt{exp}_m^{n^{O(1)}}$ time (resp., space).

In the remainder of the thesis, we also use complexity classes that are defined using alternating Turing machines. Given n the input size of problems, let $\mathrm{TA}[t(n), a(n)]$ denote the class of problems which can be solved by alternating Turing machines with running time $t(n)$ using $a(n)$ alternations.

Below are two special classes, which are both in $\mathrm{TA}[O(n), 1]$:

- Σ_2^{P}: problems that can be solved by alternating Turing machines with polynomial running time, using 1 alteration and beginning with an existential configuration. These problems are normally defined as the class of problems solvable by Turing machines in NP augmented by oracles for some coNP-complete problem, denoted $\mathsf{NP}^{\mathsf{coNP}}$.

- Π_2^{P}: problems that can be solved by alternating Turing machines with polynomial running time, using 1 alternation and beginning with a universal configuration. These problems are normally defined as the class of problems solvable by Turing machines in coNP augmented by oracles for some NP-complete problem, denoted $\mathsf{coNP}^{\mathsf{NP}}$.

In some of our complexity proofs, we use the $\forall\exists$-3CNF problem. An instance of the $\forall\exists$-3CNF problem consists of two tuples $\bar{x} = (x_1, ..., x_m)$ and $\bar{y} = (y_1, ..., y_n)$ of boolean variables and a 3CNF formula $\bar{\alpha} = \alpha_1 \wedge ... \wedge \alpha_p$, where each clause α_i is a disjunction of exactly three literals (i.e., variables from $\{x_1, ..., x_m, y_1, ..., y_n\}$ or their negations), denoted α_i^1, α_i^2, and α_i^3. The problem consists of deciding whether for every assignment $\theta_{\bar{x}}$ for $\bar{x}$, there exists an assignment $\theta_{\bar{y}}$ for $\bar{y}$ that satisfies $\bar{\alpha}$ (shortly, $\theta_{\bar{x}}\theta_{\bar{y}} \models \bar{\alpha}$). From Cook's Theorem, it follows that the $\forall\exists$-3CNF problem is Π_2^{P}-complete.

Our exposition of λ-calculus is mostly self-contained, but for background on simply typed λ-calculus see [GTL89, Hin08].

Chapter 4

Higher-Order Queries

This chapter defines the higher-order query languages which are the subject of the thesis. We will also define a particularly simple expressively equivalent subset of the language – the normal-form queries. First we define a higher-order query language, called HO, for relational databases. Then we define another one, called XQH, for XML databases, which is notable because recent proposals from the World Wide Web consortium propose adding support for higher-order functions within the XQuery standard. Lastly, we define HOCV, a higher-order complex-valued language, and show the reducibility between XQH and HOCV.

4.1 Higher-order languages for relational databases

We begin by defining HO, a higher-order language over Relational Algebra and database relations.

4.1.1 HO: A higher-order language over Relational Algebra

The syntax of the higher-order extension HO of Relational Algera is defined as in Figure 4.1. In the syntax of HO, X is a variable name, R is relation name, c is a condition, a, b are attribute names, and A is a set of attribute names.

We now formally define the types and semantics for the language.

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, $\mathcal{T} = (a_1, \ldots, a_m)$. A tuple of a relational type is (as usual)

$$
\begin{array}{rcl}
\mathit{query} & := & \mathit{const} \mid X \\
& \mid & \mathit{query} \cup \mathit{query} \mid \mathit{query} \bowtie \mathit{query} \mid \mathit{query} \setminus \mathit{query} \\
& \mid & \lambda X.\mathit{query} \mid \mathit{query}(\mathit{query}) \\
\\
\mathit{const} & := & R \mid \sigma_c \mid \pi_A \mid \bowtie \mid \rho_{a/b} \mid \cup \mid \setminus
\end{array}
$$

Figure 4.1: Syntax of the higher-order query language HO

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 $a_1 \ldots a_m$, 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 $\mathcal{T} \to \mathcal{T}'$ is a type whose domain is $(\mathcal{D}')^{\mathcal{D}}$, the set of functions from $\mathcal{D}$ to $\mathcal{D}'$, and whose order is $\texttt{order}(\mathcal{T} \to \mathcal{T}') = \max\big(\texttt{order}(\mathcal{T}) + 1, \texttt{order}(\mathcal{T}')\big)$. We abbreviate a type of the form $\mathcal{T}_1 \to ... \to \mathcal{T}_m \to \mathcal{T}'$ as $(\mathcal{T}_1 \times ... \times \mathcal{T}_m) \to \mathcal{T}'$ (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

instances 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 RA^+ to denote the operators of positive Relational Algebra, which contains the following constants: the unary selecting operator σ_c, the unary projection operator π_A, the binary join operator $\bowtie$,

    the unary renaming operator $\rho_{a/b}$, the binary union operator $\cup$. Since the semantics of these operators has been given in Section 3.1, we only give the types for them. Let $\mathcal{T} = (a_1, \ldots, a_n)$ and $\mathcal{T}' = (a'_1, \ldots, a'_m)$ be two relational types. The operator $\rho_{a_i/b}$ is of type $\mathcal{T} \to (a_1, \ldots, a_{i-1}, b, a_{i+1}, \ldots, a_n)$. Assuming $A = (c_1, \ldots, c_k)$ with $\forall i. c_i \in \{a_1, \ldots, a_n\}$, the operator π_A is of type $\mathcal{T} \to (c_1, \ldots, c_k)$. The operator σ_c is of type $\mathcal{T} \to \mathcal{T}$. The operator $\bowtie$ is of type $\mathcal{T} \to \mathcal{T}' \to (c_1, \ldots, c_k)$ with $\{c_1, \ldots, c_k\} = \{a_1, \ldots, a_n\} \cup \{a'_1, \ldots, a'_m\}$. $\cup$ is of type $\mathcal{T} \to \mathcal{T} \to \mathcal{T}$.
  - $\mathsf{CQ_C}$ denotes the signature that consists of the relational constants and the operators from which conjunctive queries can be built: the four families of operators $\rho_{a/b}$, π_A, σ_c, and $\bowtie$.
  - RA extends RA^+ with the usual difference operator $\setminus$, again parameterized by a given relational type.
  - We also consider a signature $\mathsf{RA_x}$ whose query constants include the operators π_B for every finite set of attributes B, σ_c for each condition c, X, $\cup$, 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 $(a_1 \ldots a_n)$ 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 $(a_1 \ldots a_k)$. 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 $(a_{n+1} \dots a_{n+k})$, 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 $\mathsf{RA}_\times$ will be studied for the weakly-typed calculus in Section 7.3, 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 operator, `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 $(\mathcal{T}_1 \times \dots \mathcal{T}_m \times \mathcal{T}'_1 \times \dots \mathcal{T}'_n) \to \mathcal{T}_i$.

  Then we have an operator `ifp` having type $\mathsf{U}_1 \times \dots \times \mathsf{U}_m \times \mathcal{T}'_1 \times \dots \times \mathcal{T}'_n \to \mathcal{T}_1$. The output of `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^1_j, where the doubly-indexed sequence of instances R^i_j for $i \leq m$ and j a number is formed as follows: $R^i_0 = \emptyset$ for all $i \leq m$ and $R^i_{j+1} = Q_i(R^1_j, \dots R^m_j, I_1 \dots I_n) \cup R^i_j$.

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 or variable is a term of its type;
- if X is a variable of type $\mathcal{T}$ and ρ is a term of type $\mathcal{T}'$, then $\lambda X.\, \rho$ is a term of type $\mathcal{T} \to \mathcal{T}'$;
- if τ is a term of type $\mathcal{T} \to \mathcal{T}'$ and ρ is a term of type $\mathcal{T}$, then $\tau(\rho)$ is a term of type $\mathcal{T}'$.

When $\mathcal{F}$ is RA, the syntax of the terms becomes the one introduced at the beginning of the section. The syntax for the other fragments of RA can be easily obtained from that.

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

The *order* of a term τ is the order of its type.

The *degree* of τ is the maximum order of its variables.

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 π_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.

Normal forms.

We recall the notions of β-reduction, η-expansion, and η-long β-normal form from [Hin08]. The reduction is called β because of historical reasons. *β-reduction* is the application, in any given context, of the following rewriting rule (renaming of bound variables may be necessary in order to avoid variable capture):

$$(\lambda X.\Phi)(\varphi) \quad \rightsquigarrow \quad \Phi[X/\varphi].$$

The lefthandside term above is called a *redex*. A term is said to be in *β-normal form* if it contains no redex (and hence no β-reduction can be applied to it).

Another useful transformation is that of *η-expansion*, which transforms a subterm Φ of functional type $\mathcal{T} \to \mathcal{T}'$ to the subterm $\lambda X.\, \Phi(X)$, where X is a fresh variable of type $\mathcal{T}$. In order to guarantee termination, the operation of η-expansion is restricted to the subterms Φ that do not start with the abstraction operator λ and that have no explicit argument in their context (e.g., η-expansion is never applied to the subterms Φ when they occur in a context like $\Phi(\varphi)$). For example, by applying η-expansion to a term π_A, we obtain a term $\lambda R.\pi_A R$. A term is said to be in *η-long β-normal form* (hereafter, simply *normal form*) if no β-reduction nor η-expansion (as restricted before) is possible.

Since the operations of β-reduction and η-expansion are confluent and always terminating (on well-typed terms) [GTL89], we have that every term Φ has a *unique* normal form up to renaming of variables, denoted $\Phi^{\downarrow}$. Moreover, the normal form of a term can be obtained by first applying all β-reductions and then all η-expansions. This also shows that the normal form of any term Φ of order 2 can be written as follows:

$$\Phi^{\downarrow} \;=\; \lambda Q_1 ... \lambda Q_m.\; \lambda R_1 ... \lambda R_n.\; \varphi$$

where $Q_1, ..., Q_m$ are order 1 query variables, $R_1, ..., R_n$ are relational variables, and φ is a term of order 0 with free variables among $Q_1, ..., Q_m$, $R_1, ..., R_n$, but with no occurrence of λ-abstraction. In particular if Φ is a closed term of relational type, then the normal form is just a term of relational type built up from constants, which can then be evaluated, using the semantics of the constants to get a relation. Thus we have a (naive but) effective way of evaluating closed terms. Since the size of the term in normal form is hyperexponential in the size of the input term, evaluating a higher-order term by first reducing it to its normal form is not an efficient way. In Chapter 5, we will present better evaluation techniques which combine normalization with database query evaluation.

The term hierarchy.

We introduce some notation that will be used through the rest of the thesis.

Definition 3. *Let $\mathcal{F}$ be a generic signature and let m, n be two nonnegative integers. We denote by*

- $\mathsf{HO}^n_m[\mathcal{F}]$ *the class of all closed terms of order m and degree n that are built up from constants in the signature $\mathcal{F}$ using abstraction and application,*
- $\mathsf{HO}^n[\mathcal{F}]$ *the class of all closed terms of degree at most n that are built up from constants in the signature $\mathcal{F}$ using abstraction and application,*
- $\mathsf{HO}^{\downarrow}_m[\mathcal{F}]$ *the subclass of $\mathsf{HO}^n_m[\mathcal{F}]$ consisting only of terms in normal form (note that $n = m - 1$ for terms in normal form).*

As an example, $\mathsf{HO}^0_0[\mathsf{RA}^+]$ (resp., $\mathsf{HO}^0_0[\mathsf{CQ_C}]$) is the class of all closed terms of relational type (e.g., $\Phi = (\lambda R.\, R \bowtie \rho_{a/b}(R))\{t_0, t_1\}$) that are built up from the operators of the positive Relational Algebra (resp., from the operators $\rho_{a/b}$, π_A, σ_c, and $\bowtie$) via application and abstraction over variables of order 0. Note that normal forms of terms of order 1 are the same as simple terms; hence the class $\mathsf{HO}^{\downarrow}_1[\mathsf{RA}^+]$ coincides exactly with what we have called RA^+ above, and similarly for RA, RA^+, $\mathsf{CQ_C}$ – we will thus use these notations interchangeably. We will also use UCQ to denote the simple terms (or, equivalently, order 1 terms in normal form) that are built up from the signature RA^+ by only using singleton relational constants and by allowing the union operator to appear only at the topmost level. Such a class translates efficiently to Unions of Conjunctive Queries.

Example 7. *Let R, R' be two variables of relational type $\mathcal{R} = (a, b)$, and Q be a variable of query type $\mathcal{R} \to \mathcal{R}$. We consider an order 1 degree 1 term below.*

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

We can see that τ_1 is of type $\mathcal{R} \to \mathcal{R}$ and that it returns all the pairs of nodes having a path of length 4 between them. In general, using variables of order 1, we can find paths of length doubly exponential in the input size.

Semantics.

Our semantics is simply the standard denotational semantics of the λ-calculus with an interpretation for the relational constants and the query constants. Below, we define such a semantics by exploiting an induction on the order of terms.

In order to do that, we need to first fix an interpretation for the constants and the variable domains. Formally, an interpretation $\mathcal{I}$ for the signature $\mathcal{F}$ is a function that maps (i) every constant $\texttt{const} \in \mathcal{F}$ to its semantics $[\![\texttt{const}]\!]_{\mathcal{I}}$ (e.g., $[\![\cup]\!]_{\mathcal{I}}$ is the function that maps a pair of relations $\texttt{R}_1$ and $\texttt{R}_2$ to their union $\texttt{R}_1 \cup \texttt{R}_2$) and (ii) every variable $X \in \mathcal{X}$ to its domain $Dom_{\mathcal{I}}(X)$ (e.g., if X is an order 1 query variable, then $Dom_{\mathcal{I}}(X)$ can be the set of all queries of the positive Relational Algebra). Below, we make the underlying interpretation $\mathcal{I}$ explicit by denoting the semantics of a term Φ by $[\![\Phi]\!]_{\mathcal{I}}$.

For every term Φ of the form $\texttt{const}(\varphi_1, ..., \varphi_k)$, where $k \in \mathbb{N}$ is the arity of the constant $\texttt{const} \in \mathcal{F}$, we denote by $[\![\Phi]\!]_{\mathcal{I}}$ the relation $[\![\texttt{const}]\!]_{\mathcal{I}}\big([\![\varphi_1]\!]_{\mathcal{I}}, ..., [\![\varphi_1]\!]_{\mathcal{I}}\big)$.

Given a term Φ of the form $\lambda X.\, \varphi$, we denote by $[\![\Phi]\!]_{\mathcal{I}}$ the function that maps every object x in $Dom_{\mathcal{I}}(X)$ to the object $[\![\varphi]\!]_{\mathcal{I}[X/x]}$, where $\mathcal{I}[X/x]$ is the interpretation for the extended signature $\mathcal{F} \cup \{x\}$ obtained from $\mathcal{I}$ by letting $[\![x]\!]_{\mathcal{I}[X/x]} = x$ be the interpretation for the new constant x.

Finally, given a term Φ of the form $\varphi_1(\varphi_2)$, we denote by $[\![\Phi]\!]_{\mathcal{I}}$ the object $[\![\varphi_1]\!]_{\mathcal{I}}\big([\![\varphi_2]\!]_{\mathcal{I}}\big)$.

In addition to the denotational semantics above, we will define the semantics of HO by a set of reduction rules. In the rules, $\to$ to denote a direct reduction, and $\Rightarrow$ a derivation from a sequence of $\to$ transitions. The notation of type is given after ":", and by default R, R_1, R_2 denote relational constants, which are sets of tuples. The following six rules are for query constants.

$$\frac{q \Rightarrow R \qquad R : \mathcal{T}}{\sigma_c(q) \to \{t \mid t \in R \text{ and } t \text{ satisfies } c\} : \mathcal{T}}$$

$$\frac{q \Rightarrow R \qquad R : (A) \qquad \{B\} \subseteq \{A\}}{\pi_{\{B\}}(q) \to \{t[B] \mid t \in R\} : (B)}$$

where $t[B]$ is a subtuple of t that occurs in the attribute set $\{B\}$.

$$\frac{q \Rightarrow R \quad R : (a_1, \ldots, a_m)}{\rho_{a_i/b}(q) \to \{t \mid t \in R\} : (a_1, \ldots, a_{i-1}, b, a_{i+1}, \ldots, a_m)}$$

$$\frac{q_1 \Rightarrow R_1 \qquad q_2 \Rightarrow R_2 \qquad R_1 : (A, B) \qquad R_2 : (B, C) \qquad \{A\} \cap \{C\} = \emptyset}{q_1 \bowtie q_2 \to \{\langle t_1, t_2[C]\rangle \mid t_1 \in R_1, t_2 \in R_2, t_1[B] = t_2[B]\} : (A, B, C)}$$

where A, B, C are sets of attribute names, $t_1[B]$ is a subtuple of t_1 that occurs in the attribute set $\{B\}$, and similarly for $t_2[B]$, $t_2[C]$.

$$\frac{q_1 \Rightarrow R_1 \quad q_2 \Rightarrow R_2 \quad R_1 : \mathcal{T} \quad R_2 : \mathcal{T}}{q_1 \cup q_2 \to R_1 \cup R_2 : \mathcal{T}} \qquad \frac{q_1 \Rightarrow R_1 \quad q_2 \Rightarrow R_2 \quad R_1 : \mathcal{T} \quad R_2 : \mathcal{T}}{q_1 \setminus q_2 \to R_1 \setminus R_2 : \mathcal{T}}$$

Lastly, the following rule is for β-reduction.

$$\frac{X : \mathcal{T}_2 \qquad q_1 : \mathcal{T}_1 \qquad q_2 : \mathcal{T}_2}{(\lambda X.q_1)q_2 \to q_1(X/q_2) : \mathcal{T}_1}$$

with the assumption that the names of all the λ-variables are different.

From now on, for a fixed signature $\mathcal{F}$ (e.g., $\mathcal{F} = \mathsf{RA}^+$), we tacitly assume the standard interpretation for the constants in $\mathcal{F}$ and the standard interpretation for the domains of the relational variables, which are the sets of finite relations of appropriate types. We now explain how the ordinary relational calculus embeds in our language. A term is *simple* if it contains no second-order variables and no λ-abstractions: thus, a simple term is formed by just using the constants of the signature. We identify a simple term with the query obtained by abstracting all of its relational variables and adding a fresh abstracted variable if there are none free. Under this convention RA terms correspond to Relational Algebra queries in the usual sense, RA^+ terms correspond to positive Relational Algebra queries, and CQ terms correspond to select-project-join queries [AHV95]. The signature CQ is a fragment of the signature $\mathsf{CQ_C}$ which does not include the set of relational constants. We will freely use RA, RA^+, $\mathsf{CQ_C}$, and CQ to refer to both the simple terms and the associated queries.

In contrast to the case of relation variables, we let the domains for *query variables* be unspecified a priori, and we use an auxiliary argument to completely describe their semantics. We shall denote by $\lambda\mathsf{RA}^+$ (resp., $\lambda\mathsf{CQ_C}$, $\lambda\mathsf{RA}$) the interpretation for $\mathcal{F}$ that associates with any order 1 variable Q the set of all queries of the positive Relational Algebra (resp., the set of all Conjunctive Queries with Relational Constants, the set of all Relational Algebra queries). We will sometimes refer to the range of variables as the *base*. As an example, if $\Phi = \lambda Q.\, \lambda R.\, Q(R)$, then $[\![\Phi]\!]_{\lambda\mathsf{RA}^+}$ denotes the function that maps a query $\mathtt{Q}$ of the positive Relational Algebra and a finite relation $\mathtt{R}$ to the

finite relation $\mathtt{Q}(\mathtt{R})$. Moreover, if the interpretation $\mathcal{I}$ is clear from the context, we can omit the subscript $\mathcal{I}$ from $[\![\Phi]\!]_{\mathcal{I}}$. By a slight abuse of notation, we can also write `const` in place of $[\![\mathtt{const}]\!]$ for the standard interpretation of the constant `const` in the signature $\mathcal{F}$.

Construction trees.

Tree and graph representations of λ-terms are commonly-used in analyzing reduction strategies for λ calculus. We will use a parsed representation that we call *construction trees* for our terms. The construction trees will be used in our inductive algorithms, e.g., Algorithm 2. 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 $c(st)$.

Definition 4. *The* construction tree *of a term τ is a binary tree. Each application $\tau_1(\tau_2)$ is represented by an @ node which has the left subtree is the construction tree of τ_1, and the right subtree is the construction tree of τ_2. Each λX is represented by a node, called a λ node. Each constant or variable is represented by a node labelled with its name, called a constant node or a variable node, respectively. The tree is inductively built as in Algorithm 1.*

Algorithm 1 Building the construction tree of a term

Input: A higher-order term

Output: A construction tree

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, and labelled c (resp., x).
3: The construction tree of an abstraction $\lambda x.s$ consists of a node labelled λx with a single subtree, which is the construction tree of s.
3: 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.
4: 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.
5: The construction tree of a unary query constant $c(s)$ consists of a node c whose only subtree is the construction tree of s.

Example 8 demonstrates the construction tree of a higher-order term.

Example 8. *Let τ_0 be the following term:*

$$\tau_0 := \{\lambda R.\pi_{a,b}(\rho_{b,c}(R) \bowtie \rho_{a,c}(R))\}\{(\lambda R'.\sigma_{a=3}(R'))D_0\}$$

Using Algorithm 1, we build the construction tree of τ_0 in Figure 4.2.

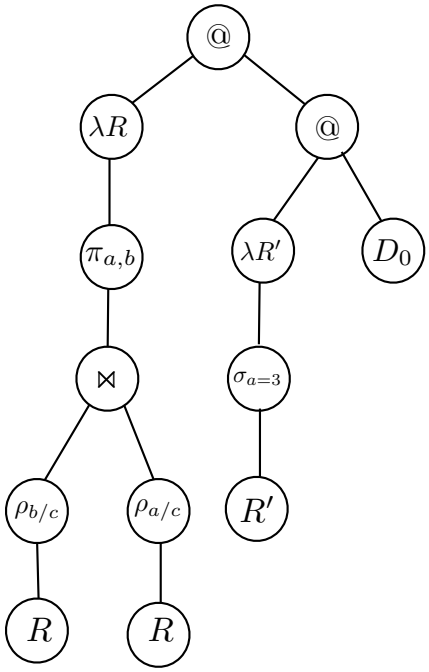


Figure 4.2: The construction tree of τ_0.

4.1.2 Expressiveness and succinctness of HO

From the definition of HO terms in normal form, we know that $\mathsf{HO}_1^{\downarrow}[\mathsf{RA}^+]$ (resp., $\mathsf{HO}_1^{\downarrow}[\mathsf{RA}]$) is the class of RA^+ (resp., RA) expressions. We now show that degree 0 terms which are not in normal form are actually familiar objects in database querying.

Proposition 4.1. *There are polynomial translations between:*

- $\mathsf{HO}_1^0[\mathsf{RA}^+]$ *and Nonrecursive Datalog*
- $\mathsf{HO}_1^0[\mathsf{RA}]$ *and Nonrecursive Datalog with Stratified Negation*
- $\mathsf{HO}_1^0[\mathsf{CQ}]$ *and Nonrecursive Datalog in which every intensional predicate occurs on the lefthandside of at most one rule.*

Proof. We start by giving the translation for the first item. Fix S the signature of a Nonrecursive Datalog and C its constants. Let $\mathcal{T}_{RA\to Datalog}$ be a polynomial reduction from an RA^+ expression to a Nonrecursive Datalog, and $\mathcal{T}_{Datalog\to RA}$ a reduction from a Nonrecursive Datalog to an RA^+ expression. The constructions of these reductions have been built, for example the one in [AHV95]. Using $\mathcal{T}_{RA\to Datalog}$, we build $\mathcal{T}_{HO^*\to Datalog}$ that translates from higher-order terms without variables that are applied to Nonrecursive Datalog. Given an input term u, we first remove all the λx such that λx is a subterm of u, then apply $\mathcal{T}_{RA\to Datalog}$ on this new term.

Algorithm 2 describes a polynomial reduction from a HO_1^0 term τ to a set $\mathcal{P}$ of Nonrecursive Datalog rules by reduction on the structure of τ. We travel bottom-up on the construction tree ξ of τ: It is easy to see that the algorithm halts within

Algorithm 2 Translation from a higher-order term to a Nonrecursive Datalog

Input: A higher-order term τ
Output: A set $\mathcal{P}$ of Nonrecursive Datalog rules

1: $\mathcal{P} := \emptyset$
2: Initialize $\mathcal{V}$ as the set of variables in τ that are either free or never applied.
3: Build the construction tree ξ for the input term τ
4: $tempHead := \emptyset$
5: **while** $\xi \neq tempHead$ **do**
6: **if** (t is the maximal subtree of ξ that all variables in t are in $\mathcal{V}$) **then**
7: convert t to a set of Datalog rules $tempRules_t$ with the goal predicate $tempHead$ and the body $\mathcal{T}_{HO^*\to Datalog}(t)$
8: Replace t in ξ by a node with label $tempHead$.
9: **end if**
10: **if** (there is a subtree $\lambda R.u(tempHead)$ in ξ) **then**
11: replace $\lambda R.u(tempHead)$ in ξ by u
12: replace all $tempHead$ in $tempRules_t$ by R
13: $S := S \cup tempRules$
14: $\mathcal{V} := \mathcal{V} \cup R$.
15: **end if**
16: **end while**
 {*At this step G contains only one node with label $tempHead$*}
17: Replace $tempHead$ in $tempRules$ by $Goal$
18: $S := S \cup tempRules$. {*This completes the construction*}
19: **return** S

polynomial time because in each step a λ node is eliminated.

The translation from a Nonrecursive Datalog program $\mathcal{P}$ to a higher-order term τ is described in Algorithm 3.

Algorithm 3 Translation from a Nonrecursive Datalog to a higher-order term

Input: A set $\mathcal{P}$ of Nonrecursive Datalog rules
Output: A higher-order term τ

1: Find $\mathcal{P}_{Goal}$ the set of rules with the head predicate $Goal$.
2: Initialize $\tau := \mathcal{T}_{Datalog \to RA}(\mathcal{P}_{Goal})$
3: Initialize $AppliedVar$ as the set of intensional predicate names in τ
4: **while** $AppliedVar \neq \emptyset$ **do**
5: Remove X from $AppliedVar$
6: Find $Body_X$ the set of rules in $\mathcal{P}$ that have head predicate X
7: $\tau := \lambda X.\tau(\mathcal{T}_{Datalog \to RA}(Body_X))$
8: Add all intensional predicate names in $Body_X$ to $AppliedVar$
9: **end while**
10: **for** Each input predicate Y that occurs in τ **do**
11: $\tau := \lambda Y.\tau$
12: **end for**
13: **return** τ

We can easily extend the translations above to the case of RA. An $\mathsf{HO}_1^0[\mathsf{RA}]$ subtree without λ nodes can be translated to an RA expression. Additionally, it is known that there is a translation between an NRDL$^\neg$ query and an RA expression.

Similarly, for the case of CQ and CQ_C, when translating HO to Datalog, we do not have two rules with the same head. In addition, when translating Datalog to HO, we do not require the Union operator. □

The following example demonstrates the translation between degree 0 terms and Datalog queries.

Example 9. *Let R, R_1, and R_2 be three relational variables respectively of type (b, c), (a, b), and (a, b, c). We consider an order 1 degree 0 term below.*

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

Let R'_1, R'_2 and R' be three new predicates, which have the same arities as R_1, R_2 and R, respectively. The term τ_3 is equivalent to the following NRDL query.

$$\begin{aligned} \mathsf{Goal}(x) &\leftarrow R'_1(x, y), R'(y, 5) \\ R'(y, z) &\leftarrow R'_2(x, y, z) \end{aligned}$$

where R'_1, R'_2 are two input predicates and R' is an intensional predicate.

Now we look at the succinctness of HO terms. We start by explaining that sharing of subterms can make unnormalized terms much more succinct than their normalized

counterparts. From a standard argument in functional programming (similar results occur in the context of Nested Relational Algebra and functional query languages, see e.g., [Koc06]) one can see that terms that use query and relation variables are much more succinct than simple RA^+-terms. What is less well-noted, perhaps, is that the same holds for degree 0 terms with respect to "flat" unions of conjunctive queries. That is:

Proposition 4.2. *There are terms $\Phi_n \in \mathsf{HO}^1_1[\mathsf{CQ}]$ (i.e. using query variables but evaluating to a query) of size $O(n)$ where any equivalent RA^+-query is of size at least 2^{2^n}.*
There are terms in $\mathsf{HO}^0_1[\mathsf{RA}^+]$ such that any equivalent union of conjunctive queries is of size at least 2^{2^n}.

Proof. As for the first part, we start by showing that $\mathsf{HO}^1_1[\mathsf{CQ}]$ terms can check the existence of paths of length 2^{2^n} in the directed graph represented by a given binary relation R. As in Example 4 on Page 6, we define a query $Q_0 = \tau_p^2$ which returns all paths of length 2 in R. Extending the argument in Example 4, we inductively define the query τ_n below.

$$\begin{aligned} \tau_n &= \lambda Q_{n-1}.\lambda R.Q_{n-1}(Q_{n-1}(R))\tau_{n-1} \\ &\dots \\ \tau_0 &= \lambda R.Q_0(Q_0(R)) \end{aligned}$$

Note that τ_0 applies two nestings of Q_0 on a relation R. Additionally, for each $i \in [1, \dots, n]$, τ_i applies two nestings of τ_{i-1} on a relation. Thus, the term τ_n takes as input a graph and returns a graph containing all the pairs of nodes having a path of length 2^{2^n} between them. Therefore, we can use a $\mathsf{HO}^1_1[\mathsf{CQ}]$ term to check for the existence of a path of length 2^{2^n} in a given binary relation R.

We now show that any RA^+-query with less than 2^{2^n} variables cannot check this by the following lemma.

Lemma 1. *Given a binary relation R, an existential First-Order logic formula with less than N variables cannot check for the existence of a path of length N in R.*

The lemma is shown using an one-sided Ehrenfeucht-Fraïsse game argument [Pez99]. Let R_1 and R_2 be binary relations which consist of linear paths of length $N-1$ and N, respectively; specifically, $R_1 = \{(u_0, u_1), \dots, (u_{N-2}, u_{N-1})\}$ and $R_2 = \{(v_0, v_1), \dots, (v_{N-1}, v_N)\}$. We need to show that for any existential First-Order logic formula f with at most $N-1$ variables, if R_2 satisfies f then R_1 also satisfies f.

The spoiler in the one-sided Ehrenfeucht-Fraïsse game will be only on R_2. We give a simple strategy for the duplicator to win the game. When the spoiler starts at any node in R_2, the duplicator always chooses to start at u_0. This makes f cannot distinguish between R_1 and R_2.

From the lemma above and the fact that any RA^+-query is linearly reducible to an existential First-Order logic formula, we know that any RA^+-query with less than 2^{2^n} variables cannot check for the existence of a path of length 2^{2^n} in a given binary relation R.

As for the second part, let A and B be two unary predicates with integer attributes (a) and (b), respectively; let R be a binary predicate with integer attributes (a, b). Let Φ_n be a query term of degree 0 that checks whether the graph represented by the binary relation R contains a path of length 2^n consisting of nodes satisfying $A \vee B$. To define Φ_n, we first write a term which returns pairs of nodes in R such that the nodes satisfy $A \vee B$:

$$R_1 = R \bowtie (\rho_{a/b} A \cup B) \bowtie (A \cup \rho_{b/a} B)$$

Then by applying n nestings of the term Q_0 defined above on R_1, we have:

$$\Phi_n = Q_0(Q_0(\ldots Q_0(R \bowtie (\rho_{a/b} A \cup B) \bowtie (A \cup \rho_{b/a} B)))$$

where the number of occurrences of Q_0 in Φ_n is n.

Now, consider a UCQ Φ'_n equivalent to Φ_n. Each disjunct D_i in Φ'_n consists of a collection of existentially quantified variables $\vec{x}$ followed by a conjunction C_i. Note that for any path π of size 2^n, there is a model R_π that has an isomorphic copy of that path and no other path of this size. For every such path π, let D_π be the disjunct that is satisfied in the corresponding model. Clearly, any two non-isomorphic paths π and π' have distinct corresponding disjuncts D_π and $D_{\pi'}$. This shows that any UCQ Φ'_n equivalent to Φ_n contains doubly exponentially many disjuncts. □

We generalize the first part of the proposition above to terms of higher-degree. Here we use the notation exp^n_m defined in Section 3.2.

Proposition 4.3. *There are terms $\Phi_n \in \mathsf{HO}^k_1[\mathsf{CQ}]$ (i.e. using query variables but evaluating to a query) of size $O(n)$ where any equivalent RA^+-query is of size at least exp^n_{k+1}.*

Proof. We use $\mathtt{Q}$ and R from the proof of Proposition 4.2. From the coding in Proposition 5.12, we can use a degree k term to represent a term of degree 1 of the form $\lambda R.\,[\mathtt{exp}_k^n](\mathtt{Q})(R)$. This term can check for the existence of a path of length $2^{\mathtt{exp}_k^n}$, which is equal to $\mathtt{exp}_{k+1}^n$, in R. Using Lemma 1 on Page 36, we have that any RA^+-query with less than $\mathtt{exp}_{k+1}^n$ variables cannot check this. □

4.2 Higher-order languages for XML databases and complex values

4.2.1 XQH: A higher-order extension of Core XQuery

In this work, we only consider unranked labeled ordered trees without fixing the values of labels. Additional XML features, such as attributes, can be coded into this extension. This coding does not impact evaluation, although it does have implications for typing – see Section 7.2. We maintain the distinction between data and queries in XQH. The syntax of the higher-order extension XQH of Core XQuery defines queries as in Figure 4.3.

$$
\begin{array}{rcl}
\mathit{query} & := & () \mid \langle a \rangle \mathit{query} \langle / a \rangle \mid \mathit{query}\ \mathit{query} \\
& \mid & \mathit{var} \mid \mathit{var}/\mathit{axis} :: \nu \\
& \mid & \text{for } \mathit{var} \text{ in } \mathit{query} \text{ return } \mathit{query} \\
& \mid & \text{if } \mathit{cond} \text{ then } \mathit{query} \\
& \mid & [\mathcal{V}]\mathit{query} \\
& \mid & \mathit{query}(\mathit{query}) \\
\mathcal{V} & := & \emptyset \mid \mathit{var}, \mathcal{V} \\
\mathit{cond} & := & \mathit{query} = \mathit{query} \mid \mathit{query} \mid \mathit{not}(\mathit{cond})
\end{array}
$$

Figure 4.3: Syntax of the higher-order query language XQH

In the syntax of XQH, a denotes an XML tag, axis denotes XPath paths, ν denotes node tests, and var is a set of XQuery variables. The equivalence $=$ can be (a) atomic equality (denoted $[=_{\mathit{atomic}}]$), which compares labels of two leaves, or (b) deep equality (denoted $[=_{\mathit{deep}}]$), an isomorphism test of two nodes (identified with their subtrees). Let XQHP denotes the positive fragment of XQH, which is the fragment with atomic equality and without $\mathit{not}(\mathit{cond})$.

The syntax of XQH is based roughly on that of Koch's Core XQuery (reviewed in Section 4.2.3), adding support for higher-order operators. It adds the general λ-abstraction construct $[\mathcal{V}]query$, in which $\mathcal{V}$ is a list of variables. For simplicity, $[\mathcal{V}_1][\mathcal{V}_2]query$ is rewritten as $[\mathcal{V}_1, \mathcal{V}_2]query$ and $[]query$ is rewritten as $query$. The construct $query(query)$, which we sometimes write $query$ @ $query$ for convenience of parsing, denotes the application of a query to another query.

Similarly to Core XQuery, we do not represent other XQuery operators, e.g., *let*, *true*, *and*, *or*, because they can be derived from the operators above. For example, one can code *true* using a query that always evaluates to a nonempty collection. For more details on how to encode these operations, see [Koc06].

Before giving the semantics of XQH, we inductively define a set of *abstract types*, called AT, and their *order* as follows.

- $\mathcal{B}$ is the "base type", representing any function-free object; $\texttt{order}(\mathcal{B}) = 0$.
- If $\mathcal{T}_1$ and $\mathcal{T}_2$ are AT, then $\mathcal{T}_1 \to \mathcal{T}_2$ is an AT and

$$\texttt{order}(\mathcal{T}_1 \to \mathcal{T}_2) = \max\big(\texttt{order}(\mathcal{T}_1) + 1, \texttt{order}(\mathcal{T}_2)\big)$$

Each abstract type has a *denotation*; the base type maps to sequences of nodes within unranked labeled ordered trees. All the trees and lists of trees are associated with the basic abstract type $\mathcal{B}$. $\mathcal{T}_1 \to \mathcal{T}_2$ denotes the set of functions from the denotation of $\mathcal{T}_1$ to the denotation of $\mathcal{T}_2$.

For each AT we assume a set of variables associated with that AT. We also give type inference rules that assign abstract types to each XQH query. We give the rules for several cases; the rules for the other cases can be inferred from the syntax of the queries.

- AT of each query without an abstraction is $\mathcal{B}$.
- AT of a query with a nonempty abstraction [] is defined as: $\mathsf{AT}([x, \mathcal{V}]query) = \mathsf{AT}(x) \to \mathsf{AT}([\mathcal{V}]query)$
- If $\mathsf{AT}(query_1) = \mathcal{T}_1 \to \mathcal{T}_2$ and $\mathsf{AT}(query_2) = \mathcal{T}_1$, then $\mathsf{AT}(query_1 \text{ @ } query_2) = \mathcal{T}_2$.

For example, the expression $\$x$, where $\$x$ is a variable of base type, has type $\mathcal{B} \to \mathcal{B}$: it is an ordinary query.

There are of course, queries that can not be assigned a type – e.g., if we have a subquery $[var, D]q_2$ @ q_1, with $\mathsf{AT}(var) \neq \mathsf{AT}(q_1)$. It is easy to check this (syntactic)

"well-typedness condition". Except in Chapter 7 we will always assume that queries are well-typed.

To define the semantics of the language, we can use the standard semantics of the constants together with the definition of λ-calculus in the way we define HO. However, to be complete and self-contained, here the semantics of XQH is defined by a set of reduction rules. In the rules, we use "," to denote concatenation, $\rightarrow$ to denote a direct reduction, and $\Rightarrow$ a derivation from a sequence of $\rightarrow$ transitions. Within reductions we allow an extended syntax with a constant term for every node sequence within a document. The rules include:

$$\frac{}{() \rightarrow []} \qquad \frac{q \Rightarrow \tau \qquad \mathsf{AT}(q) = \mathcal{B}}{\langle a \rangle q \langle /a \rangle \rightarrow [\langle a \rangle \tau \langle /a \rangle]}$$

$$\frac{q_1 \Rightarrow \tau_1 \qquad q_2 \Rightarrow \tau_2 \qquad \mathsf{AT}(q_1) = \mathsf{AT}(q_2) = \mathcal{B}}{q_1 q_2 \rightarrow \tau_1, \tau_2}$$

$$\frac{\mathsf{AT}(q) = \mathcal{B} \qquad q \Rightarrow \tau}{q/axis :: \nu \rightarrow [\tau_1, \ldots, \tau_m]}$$

where $[\tau_1, \ldots, \tau_m]$ is a list of τ's nodes are ordered by document order w.r.t. τ. Additionally, for every i, the root of τ_i is labeled by ν, and the path from the root of τ to the root of τ_i matches $axis$.

$$\frac{\mathsf{AT}(x) = \mathsf{AT}(q_1) = \mathcal{B}\ q_1 \Rightarrow (\tau_1, \ldots, \tau_n)\ \forall i \leq n\ q_2(x/\tau_i) \Rightarrow \tau_i'}{(\text{for } x \text{ in } q_1 \text{ return } q_2) \rightarrow \tau_1', \cdots, \tau_n'}$$

$$\frac{\mathsf{AT}(cond) = \mathcal{B} \quad cond \Rightarrow [\tau_1, \ldots]}{(\text{if } cond \text{ then } q) \rightarrow q} \qquad \frac{\mathsf{AT}(cond) = \mathcal{B} \quad cond \Rightarrow []}{(\text{if } cond \text{ then } q) \rightarrow []}$$

Above $[\tau_1 \ldots]$ is always a non-empty nodelist.

$$\frac{\mathsf{AT}(q_1) = \mathsf{AT}(q_2) = \mathcal{B} \qquad q_1 \Rightarrow \tau_1 \quad q_2 \Rightarrow \tau_2 \quad \tau_1 = \tau_2}{q_1 = q_2 \rightarrow true}$$

where $=$ can be atomic equality or deep equality as defined in the syntax of XQH.

$$\frac{[x, V]q_2 \qquad \mathsf{AT}(x) = \mathsf{AT}(q_1) \qquad q_2(x/q_1) \Rightarrow q_2'}{([x, V]q_2\ @\ q_1) \rightarrow [V]q_2'}$$

with the assumption that all the variables inside the square brackets (i.e., λ-variables) are different. Although this β-reduction rule adds non-determinism (due to choice of reduction), one can show that a unique normal form exists.

A significant difference from XQuery is that in the semantics there are no variable bindings ("dynamic environments") representing interaction with an external input document. Instead, the inputs must be hard-coded into the query, with documents

built up explicitly via node construction. We do this to keep the semantics less cluttered, and more similar to our higher-order nested relational language. We can still trivially translate every query evaluation problem into our language.

Example 10. *Let $\$D$ and $\$R$ be two variables that have* AT *equal to $\mathcal{B}$, $\$Q$ be a query variable that has* AT *equal to $\mathcal{B} \to \mathcal{B}$. The query:*

$$
\begin{aligned}
&([\$Q\ \$D]\$Q(\$Q(\$D)))\\
&[\$S]\\
&\langle SEQ\rangle\{\\
&\ for\ \$i\ in\ \$S/child :: route, \$j\ in\ \$S/child :: route\\
&where\ \$i/to = \$j/from\ return\\
&\langle route\rangle\{\langle from\rangle\{\$i/from\}\langle/from\rangle, \langle to\rangle\{\$j/to\}\langle/to\rangle\}\langle/route\rangle\\
&\}\langle/SEQ\rangle
\end{aligned}
$$

consists of a higher-order query performing composition – $[\$Q\$D]\$Q(\$Q(\$D)))$ – applied to a query that takes a sequence of "routes", where each "route" contains a pair of "from" and "to", and joins them. The composition will return routes with three intermediate legs. We can extend this idea to express an exponential number of joins succinctly using query variables – indeed, this is one cause of the high complexity we will see for higher-order queries.

The *order* of a query is the order of the AT of the query. The *degree* of an XQH term is the highest order of variables in the term. Similarly to HO, we denote XQH_n the set of XQH terms of order n, and XQH^m the set of XQH terms of degree m.

For evaluation of XQH, we will be interested in queries of order 0 – i.e. ones that evaluate to a document.

Theorem 4.4. *The operational semantics of* XQH *is sound. Specifically, when applying two different sequences of rules on an input term, we receive two terms of the same semantics and same type.*

Proof. Let τ be the input term, τ_1 and τ_2 be two terms after reduction. Using the diamond property, we can show that if τ_1 can be reduced to a term τ_3, then τ_2 is also reducible to τ_3. Thus τ_1 and τ_2 must be equivalent in semantics. The rest of the proof is given to show the equivalence in type of τ_1 and τ_2.

We prove by induction on the applications of the operational rules.

- The case $\tau = \langle a \rangle q \langle a \rangle$: If $q \Rightarrow q'$, then by the induction hypothesis, q is of the same type as q'. Thus, $\langle a \rangle q' \langle a \rangle$ is of the same type as $\langle a \rangle q \langle a \rangle$.

- The case $\tau = q_1 q_2$: If $q_1 \Rightarrow q_1'$, then by the induction hypothesis, q_1' is of the same type as q_1. Thus, $q_1' q_2$ is of the same type as $q_1 q_2$. Similarly, if $q_2 \Rightarrow q_2'$, then by induction hypothesis, q_2' is of the same type as q_2. Thus, $q_1 q_2'$ is of the same type as $q_1 q_2$.

- The case $\tau = q/axis :: \nu$: If $q \Rightarrow q'$, then by the induction hypothesis, q' is of the same type as q. Thus, $q'/axis :: \nu$ is of the same type as $q/axis :: \nu$.

- The *for* rule: Assuming the type of q_2 is $\mathcal{T}_2$, the type of *(for* x *in* q_1 *return* q_2*)* is $\{\mathcal{T}_2\}$. The type of $q_2(x/\tau_i)$ is $\mathcal{T}_2$ for all $1 \leq i \leq n$; therefore $q_2(x/\tau_1), \ldots, q_2(x/\tau_n)$ are of type $\{\mathcal{T}_2\}$.

- The case $\tau =$ if *cond* then q: The type of the output is of the same type as q, which is the same type as if *cond* then q.

- The case $\tau = q_2 \mathbin{@} q_1$:

$$\frac{[x, V]q_2 \qquad \mathsf{AT}(x) = \mathsf{AT}(q_1) \qquad q_2(x/q_1) \Rightarrow q_2'}{([x, V]q_2 \mathbin{@} q_1) \rightarrow [V]q_2'}$$

  The condition on the type of q_1 and $[x, V]q_2$ implies that x and q_1 are of type $\mathcal{T}_1$ and $[x, V]q_2$ is of type $\mathcal{T}_1 \rightarrow \mathcal{T}_2$. Using q_3 to denote $[V]q_2''$, we need to show that q_3 is of type $\mathcal{T}_2$. This is shown based on the following lemma about the substitution of x by q_2:

  Lemma 2. *Let ρ be a term of type $\mathcal{T}$ containing a free variable y, which is of the same type as another term ρ'. We have the type of $\rho[y/\rho']$ is $\mathcal{T}$.*

  The lemma is shown by induction on ρ:

  - If $\rho = y$, then $\rho[y/\rho'] = \rho'$, which is of type $\mathcal{T}$.
  - If $\rho = C(\rho_1)$ with C a constant, then $\rho_1[y/\rho']$ is of the same type as ρ_1 by the induction hypothesis. Thus $C(\rho_1[y/\rho'])$, which is $\rho[y/\rho']$, is of the same type as $C(\rho_1)$.
  - If $\rho = \rho_1(C)$ with C a constant, then $\rho_1[y/\rho']$ is of the same type as ρ_2 by the induction hypothesis. Thus $\rho_1[y/\rho'](C)$, which is $\rho[y/\rho']$, is of the same type as $\rho_1(C)$.

– If $\rho = (\rho_1)(\rho_2)$, then $\rho_1[y/\rho']$ and $\rho_2[y/\rho']$ are respectively of the same type as ρ_1 and ρ_2 by the induction hypothesis. Thus $(\rho_1[y/\rho'])(\rho_2[y/\rho'])$, which is $\rho[y/\rho']$, is of the same type as $(\rho_1)(\rho_2)$.

The soundness of all the cases shows the theorem. □

4.2.2 HOCV: A higher-order complex-valued query language

We now define a corresponding language over complex values, basing it on the complex-valued query language Monad Algebra. In general, complex values can be built on top of sets, bags, or lists. Here we give the formal definition only for the set-based version of the higher-order complex-valued language HOCV. The list-based version, named HOCV_L, extends Monad Algebra on lists in a similar way.

Nested relational 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. For simplicity, we often assume all attributes range over the integers $\mathbb{Z}$.

Next we will define the *types* along with their *order*. The basic types are the collection of attribute ranges. We extend basic types to *nested relational types* as follows. Basic types are nested relational types. If $\mathcal{T}_1, \ldots, \mathcal{T}_n$ and $A_1, \ldots, A_n$ with $n \geq 1$ are nested relational types and attribute names respectively, then $\langle A_1 : \mathcal{T}_1, \ldots, A_n : \mathcal{T}_n \rangle$ and $\{\mathcal{T}_1\}$ are nested relational types.

We manipulate nested relational types by using the standard operations on lists, such as concatenation $\mathcal{T} + \mathcal{T}'$ (assuming no overlap of $\mathcal{T}$ and $\mathcal{T}'$), adding nesting $\{\mathcal{T}\}$ of $\mathcal{T}$, and the projection $\pi_A(\mathcal{T})$, for an attribute A in $\mathcal{T}$. The denotation of a nested relational type is the collection of all finite instances over the type, where the collection of instances is defined in the obvious way.

As in [TBW92], we define a "boolean type", denoted $\{\langle\rangle\}$. Note that there are only two instances of type $\{\langle\rangle\}$, namely, the empty instance $\emptyset$, which we identify with `false`, and the singleton, also denoted $\{\langle\rangle\}$, which we identify with `true`.

Higher-order types over nested relational types.
Nested relational types are the basic building blocks of more complex types. We will introduce further types now, and the notion of order. The order of any nested relational type is 0. We define *higher-order types over nested types* by using the function type constructor: if $\mathcal{T}, \mathcal{T}'$ are types with denotation $\mathcal{D}, \mathcal{D}'$, then $\mathcal{T} \to \mathcal{T}'$ is

a type with denotation the set of functions from $\mathcal{D}$ to $\mathcal{D}'$, whose order is

$$\texttt{order}(\mathcal{T} \to \mathcal{T}') = \max\big(\texttt{order}(\mathcal{T}) + 1, \texttt{order}(\mathcal{T}')\big)$$

We abbreviate a type of the form $\mathcal{T}_1 \to \ldots \to \mathcal{T}_m \to \mathcal{T}'$ as

$(\mathcal{T}_1 \times \ldots \times \mathcal{T}_m) \to \mathcal{T}'$ (an abbreviation only, since we have no product operation on types). Similarly we will write elements of such types in their curried form. We refer to order 1 types as *query types*.

Constants.

We fix a set of constants of each type $\mathcal{T}$. Constants can be thought of as specific instances 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. The order of a constant is the order of a type. We consider a signature MA that consists of the following constants:

- Our signature includes constants for all instances, referred to as *nested relational constants*.
- We consider the following order 1 constants – i.e. *query constants*.
  1. singleton set construction of type $\mathcal{T} \to \{\mathcal{T}\}$:
  2. flatten of type $\{\{\mathcal{T}\}\} \to \{\mathcal{T}\}$:
  $$\frac{X \Rightarrow \{\tau_1, \ldots, \tau_n\}}{flatten(X) \to \tau_1 \cup \ldots \cup \tau_n}$$
  3. pairing of type $\langle A_1 : \{\mathcal{T}_1\}, \ldots, A_n : \mathcal{T}_n\rangle \to \{\langle A_1 : \mathcal{T}_1, \ldots, A_n : \mathcal{T}_n\rangle\}$:
  $$\frac{X \Rightarrow \langle A_1 : \tau_1, \ldots, A_n : \tau_n\rangle}{pairwith_{A_1}(X) \to \{\langle A_1 : \rho_1, \ldots, A_n : \tau_n\rangle \mid \rho_1 \in \tau_1\}}$$
  4. for each nested relational type containing an attribute A_i, the unary operator π_{A_i} of type $\langle A_1 : \mathcal{T}_1, \ldots, A_n : \mathcal{T}_n\rangle \to \mathcal{T}_i$:
  $$\frac{X \Rightarrow \langle A_1 : \tau_1, \ldots, A_n : \tau_n\rangle}{\pi_{A_i}(X) \to \tau_i}$$
  5. for any type $\mathcal{T}$ the binary operator $\cup$, which returns the union of two order 0 terms of type $\mathcal{T}$:
  $$\frac{X_1 \Rightarrow \tau_1 \quad X_2 \Rightarrow \tau_2}{X_1 \cup X_2 \to \tau_1 \cup \tau_2}$$

6. for each relational type $\mathcal{T}$ the unary operator $\sigma_{A_i=A_j}$, which selects a subset of the tuples from a given nested relation of type $\langle A_1 : \mathcal{T}_1, \ldots, A_n : \mathcal{T}_n\rangle$, where $=$ is either (a) "atomic equality" $[=_{atomic}]$, that is, a label comparison, or (b) $[=_{deep}]$ isomorphism on nested relations;
7. tuple formation:

$$\frac{\forall i. f_i \Rightarrow \tau_i}{\langle A_1 : f_1, \ldots, A_n : f_n\rangle \to \langle A_1 : \tau_1, \ldots, A_n : \tau_n\rangle}$$

- Lastly, we consider an order 2 constant, named *map*, of type $(\mathcal{T} \to \mathcal{T}') \to \{\mathcal{T}\} \to \{\mathcal{T}'\}$:

$$\frac{X \Rightarrow \{\tau_1, \ldots, \tau_n\} \qquad \forall i. f(\tau_i) \Rightarrow \tau_i'}{map(f, X) \to \{\tau_1', \ldots, \tau_n'\}}$$

Similarly to the notation in [Koc06], we use $flatmap$ to denote $flatten(map)$. The fragment of MA with atomic equality and without difference is called MA^+.

Terms.
Higher-order *terms* are built up from the constants above and variables of higher-order type by using the operations of abstraction and application:

- every constant or variable is a term of its type;
- if X is a variable of type $\mathcal{T}$ and ρ is a term of type $\mathcal{T}'$, then $\lambda X.\, \rho$ is a term of type $\mathcal{T} \to \mathcal{T}'$;
- τ is a term of type $\mathcal{T} \to \mathcal{T}'$ and ρ is a term of type $\mathcal{T}$, then $\tau(\rho)$ is a term of type $\mathcal{T}'$.

The positive fragment, which contains atomic equality and does not contain the difference operator, is called HOCVP.

As with XQH there are conditions for a term to be well-typed – these now include not just the compatibility rules for application, but conditions needed for the individual constants to have a valid transition – again, we omit these here, and assume that terms are well-typed. Similar to the semantics of HO terms in Subsection 4.1.1, the semantics of HOCV terms is the standard operational semantics given by the rules for constants and β-reduction. We fix an interpretation for the constants in the signature $\mathcal{F}$, which is MA or MA^+, as a function $\mathcal{I}$ that maps every constant $\texttt{const} \in \mathcal{F}$ to its semantics $[\![\texttt{const}]\!]_{\mathcal{I}}$. The semantic function $[\![\tau]\!]_{\mathcal{I}}$ is inductively defined from $\mathcal{I}$ to map each free variable in the term to an object of the corresponding type. We omit

the definition of this function because it is the same as that for HO terms in Subsection 4.1.1. As for the relational case, we denote by $\lambda\mathsf{MA}$ and $\lambda\mathsf{MA}^+$ the interpretation for MA and MA^+, respectively.

The *order* of a term τ is the order of its type. We say that a term τ is *closed* if it contains no free occurrences of variables. One can show that well-typed closed terms of order 0 evaluate under the operational semantics to a unique nested relation.

The *degree* of a HOCV term is the highest order of variables in the term. We denote HOCV^k (resp., $\mathsf{HOCV}^k_\mathsf{L}$) the fragment of HOCV (resp., HOCV_L) terms of degree k.

As in the relational case, we also define the *size* of a HOCV term inductively as follows. The size of a nested relational constant is the size of the corresponding instance. 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.

4.2.3 Expressiveness and succinctness of HOCV and XQH

We recall the correspondence between XQuery and complex-valued languages proved by Koch. Koch has considered a fragment of XQuery, named Core XQuery (or XQ for short), with abstract syntax:

$$\begin{aligned} \mathit{query} &:= () \mid \langle a\rangle \mathit{query} \langle/a\rangle \mid \mathit{query}\ \mathit{query} \\ &\mid \mathit{var} \mid \mathit{var}/\mathit{axis} :: \nu \\ &\mid \text{for } \mathit{var} \text{ in } \mathit{query} \text{ return } \mathit{query} \\ &\mid \text{if } \mathit{cond} \text{ then } \mathit{query} \\ \mathit{cond} &:= \mathit{var} = \mathit{var} \mid \mathit{query} \end{aligned}$$

From the syntax, we can see that XQ is a special case of XQH where higher-order variables are absent. On the complex-valued side, Koch considered Monad Algebra on lists in [Koc06]. This is equivalent to $\mathsf{HOCV}^0_\mathsf{L}$, which consists of HOCV_L terms without higher-order variables. Koch's result can thus be restated as saying that there exists a polynomial reduction between the evaluation problems for XQH^0 and $\mathsf{HOCV}^0_\mathsf{L}$.

We note that the correspondence extends to the higher-order setting:

Proposition 4.5. *Given $k \geq 0$, evaluating XQH^k (resp., XQHP^k) queries and evaluating $\mathsf{HOCV}^k_\mathsf{L}$ (resp., $\mathsf{HOCVP}^k_\mathsf{L}$) terms are polynomially reducible to each other.*

Proof. Koch's work [Koc06] gives mappings between complex values and data trees. The work then provides polynomial translations between Monad Algebra and XQ w.r.t. these mappings. We will keep the original data mappings and extend the translation rules in our proof.

First, we give a translation from a $\mathsf{HOCV}^k_{\mathsf{L}}$ term to an XQH^k query. This translation will also be a translation from a $\mathsf{HOCVP}^k_{\mathsf{L}}$ term to an XQHP^k query. For every order 1 term other than query variables we use the translation of Koch, which returns an XQuery expression with an additional free variable.

$$
\begin{aligned}
\mathsf{XQ}(\lambda X.\tau) &:= [X]\mathsf{XQ}(\tau)\\
\mathsf{XQ}(\tau(\rho)) &:= \mathsf{XQ}(\tau)@\mathsf{XQ}(\rho)\\
\mathsf{XQ}(\langle A_1 : f_1, \ldots, A_k : f_k\rangle) &:= [\$x]\langle tup\rangle\langle a_1\rangle\mathsf{XQ}(f_1)(\$x)\langle/a_1\rangle \ldots \langle a_k\rangle\mathsf{XQ}(f_k)(\$x)\langle/a_k\rangle\langle/tup\rangle\\
\mathsf{XQ}(\pi_i) &:= [\$x]\{\$x/a_i/*\}\\
\mathsf{XQ}(sng) &:= [\$x]\langle list\rangle\{\$x\}\langle/list\rangle\\
\mathsf{XQ}(map(f)) &:= [\$x]\langle list\rangle\{\text{for } \$y \text{ in } \$x/* \text{ return } \mathsf{XQ}(f)(\$y)\}\langle/list\rangle\\
\mathsf{XQ}(flatten) &:= [\$x]\langle list\rangle\{\$x/list/*\}\langle/list\rangle\\
\mathsf{XQ}(pairwith_i) &:= [\$x]\langle list\rangle\{\text{for } \$y \text{ in } \$x/a_i/list/* \text{ return } \langle tup\rangle\\
&\quad \langle a_1\rangle\{\$x/a_1/*\}\langle/a_1\rangle \ldots \langle a_i\rangle\{\$y\}\langle/a_i\rangle \ldots\\
&\quad \langle a_k\rangle\{\$x/a_k/*\}\langle/a_k\rangle\}\langle/tup\rangle\}\langle/list\rangle\\
\mathsf{XQ}(f \cup g) &:= [\$x]\langle list\rangle\{(\mathsf{XQ}(f)(\$x))/*\}\{(\mathsf{XQ}(g)(\$x))/*\}\langle/list\rangle\\
\mathsf{XQ}(\sigma_{A_i=A_j}) &:= [\$x]\langle list\rangle\{\text{if (some } \$y \text{ in } \$x/a_i/* \text{ satisfies}\\
&\quad \text{some } \$z \text{ in } \$x/a_j/* \text{ satisfies } (\$y = \$z)) \text{ then } \langle tup/\rangle\}\langle/list\rangle\\
\mathsf{XQ}(true) &:= [\$x]\{\text{if } \$x \text{ then } \langle yes/\rangle\}
\end{aligned}
$$

At the end of the translation, we combine subterms of the form $[X_1]\ldots[X_n]$ into $[X_1, \ldots, X_n]$ to fit the syntax of XQH.

Second, we give rules translating an XQH (resp., XQHP) query to a $\mathsf{HOCV}_{\mathsf{L}}$ (resp.,

$\mathsf{HOCVP}_\mathsf{L}$) expression.

$$
\begin{aligned}
\mathsf{MA}([X,l]\tau) &:= \lambda X.\mathsf{MA}([l]\tau)\\
\mathsf{MA}(\tau@\rho) &:= \mathsf{MA}(\tau)(\mathsf{MA}(\rho))\\
\mathsf{MA}(\alpha\beta) &:= \mathsf{MA}(\alpha)\cup\mathsf{MA}(\beta)\\
\mathsf{MA}(\langle a\rangle\alpha\langle/a\rangle) &:= \langle label : a, children : \mathsf{MA}(\alpha)\rangle \circ sng\\
\mathsf{MA}(\$x) &:= \sigma_{N=\$x}\circ map(\pi_V)\\
\mathsf{MA}(\$x/*) &:= \sigma_{N=\$x}\circ flatmap(\pi_V\circ\pi_{children})\\
\mathsf{MA}(\$x/a) &:= \sigma_{N=\$x}\circ flatmap(\pi_V\circ\pi_{children}\circ\sigma_{label=a})\\
\mathsf{MA}(\text{for } \$x \text{ in } \alpha \text{ return } \beta) &:= \langle 1:id, 2:MA(\alpha)\rangle\circ pairwith_2\circ\\
&\quad\ flatmap((\pi_1\cup(\langle N:\$x, V:\pi_2\rangle\circ sng))\circ\mathsf{MA}(\beta))\\
\mathsf{MA}(\text{if } \alpha \text{ then } \beta) &:= \langle 1:id, 2:MA(\alpha)\circ true\rangle\circ pairwith_2\circ flatmap(\pi_1\circ\mathsf{MA}(\beta))\\
\mathsf{MA}(\text{not } \alpha) &:= \mathsf{MA}(\alpha)\circ map(\langle\rangle)\circ\ \text{not}\\
\mathsf{MA}(\$x=\$y) &:= \langle 1:\sigma_{N=\$x}, 2:\sigma_{N=\$y}\rangle\circ pairwith_1\circ\\
&\quad\ flatmap(pairwith_2)\circ\sigma_{1.V=2.V}
\end{aligned}
$$

In both translations, a variable is always translated to a variable of the same order. Thus the XQH query and the HOCV term are of the same order.

□

Therefore, from now on we study the complexity of HOCV_L, from which we can derive the complexity of XQH. The next section concentrates on the results for the set-based language HOCV, but the complexity results easily carry over to the list-based version HOCV_L.

We now study the succinctness for positive fragments of HOCV and XQH. This subsection considers the expansion of the size of HOCV and XQH terms when reducing high-order variables.

Proposition 4.6. *There are terms in* HOCVP^1 *(resp.,* XQHP^1*) of size* $O(n)$ *such that any equivalent* HOCVP^0 *(resp.,* XQHP^0*) term is of size at least* 2^n*.*

Proof. Similar to the proof of Proposition 4.2, we use a nested relation D to represent a directed graph. Let the input size be n. We build an HOCVP^0 term `Q` that returns all paths of length 2^n in D.

Using variables of order 0 and 1, we can build an HOCVP^1 term $\tau := \tau_n(D)$ that iterates $\mathtt{Q}$ a doubly-exponential number of times on D.

$$\begin{aligned}\tau_n &:= (\lambda Q_1.\lambda R_1.(Q_1(Q_1(R_1))))\tau_{n-1}\\ &\cdots\\ \tau_0 &:= \lambda R.(Q_0(Q_0(R)))\end{aligned}$$

Using Lemma 1 on Page 36, we can show that positive First-Order logic expressions require 2^{2^n} variables to check the existence of a path of doubly exponential length. Additionally, an XQHP^0 expression of size less than 2^n cannot represent a positive First-Order logic expression with 2^{2^n} variables [BK09]. Therefore, we require XQHP^0 terms (and also HOCVP^0 terms) of exponential size to check the existence of paths of doubly-exponential length. □

We generalize the proposition above to the following proposition:

Proposition 4.7. *There are terms in* HOCVP^k *(resp.,* XQHP^k*) of size* $O(n)$ *such that any equivalent* HOCVP^0 *(resp.,* XQHP^0*) term is of size at least* $\mathtt{exp}_k^n$*.*

Proof. We prove the theorem for HOCVP terms. Using the translations between HOCVP and XQH in the proof of Proposition 4.5, we can get the result for XQH terms.

We use D and $\mathtt{Q}$ from the proof of Proposition 4.6 above. Using the iteration functions from Theorem 5.31 in Chapter 5, we build a term in HOCVP^k that can repeat the application of $\mathtt{Q}$ on D $\mathtt{exp}_k^n$ times. This HOCVP^k term checks the existence of a path of length $\mathtt{exp}_{k+1}^n$. From Lemma 1 on Page 36, we need a positive First-Order logic expression with $\mathtt{exp}_{k+1}^n$ variables to check the existence of a path of length $\mathtt{exp}_{k+1}^n$. In addition, an XQHP^0 expression (and also an HOCVP^0 expression) of size less than $\mathtt{exp}_k^n$ can not represent a positive First-Order logic expression with $\mathtt{exp}_{k+1}^n$ variables [BK09]. Therefore, we require HOCVP^0 terms of size $\mathtt{exp}_k^n$ to check the existence of paths of length $\mathtt{exp}_{k+1}^n$. □

Chapter 5

Evaluation of Higher-Order Terms

This chapter gives 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 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 fixed point operators. We also study evaluation of the higher-order extensions for complex values and for XML databases.

We start with higher-order terms of "degree 0": those that have variables ranging only over databases. We then extend to the first higher-order case: "degree 1", 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 β-reduction with the results on degree 0. Building on the degree 1 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 higher-order queries.

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.

Similarly, we show the complexity of evaluating HOCV terms containing variables of arbitrary order. We also isolate subclasses with lower complexity. We use the complexity results for HOCV and correspondence between XQH and HOCV to get complexity bounds on the higher-order language for XQuery XQH.

5.1 The evaluation problem

Definition 5 (The evaluation problem)**.** *The* evaluation problem *takes as input a well-typed term τ of order 0, along with an order 0 constant D. The output is true iff the evaluation of τ contains D.*

When the input term is of boolean type, the output is true iff the term evaluates to the (unique) nonempty instance of the boolean type.

The computational complexity of the evaluation problem will be considered in terms of the *size* of the terms, as defined earlier in Subsection 4.1.1 and Subsection 4.2.2.

We use the definition above when considering the *combined complexity* of the problem, where the size of the input is the database size plus the term size. To simplify the proofs in the next sections, we study the evaluation problem for boolean queries. The evaluation problem in Definition 5 is polynomially reducible to the evaluation problem for boolean queries as follows. We show this for the relational case; it is similar for the cases of XML and complex values. The evaluation of τ contains all the tuples in D iff $\bigwedge_{t \in D} \pi_\emptyset(\tau \ltimes \{t\}) = \texttt{true}$. Thus, we have reduced the evaluation problem for the non-boolean term τ above to the evaluation problem for the boolean term $\bigwedge_{t \in D} \pi_\emptyset(\tau \ltimes \{t\})$.

To consider *query complexity*, where the database is fixed, and *data complexity*, where the query is fixed, we need to separate query and data as in the following definition.

Definition 6 (The evaluation problem for order 1 terms)**.** *The* evaluation problem for order 1 terms *takes as input a well-typed term of order 1, along with a set $\mathcal{C}$ of order 0 constants and an order 0 constant D. The output is true iff the evaluation of the application of the term on the constant set $\mathcal{C}$ contains D.*

We will show that the data complexity, where the higher-order term is fixed, of our higher-order query languages is in P. By applying β-reduction, we reduce the input term to a lower-order term which is equivalent to an NRDL query, or a Monad Algebra expression, or a Core XQuery expressions. Even though the size of the term increases, the complexity of evaluation is not affected because the term is fixed. Thus, the data complexity of higher-order query languages is the same as that for ordinary query languages. We know that the data complexity of NRDL, Monad Algebra, and Core XQuery is in P [Var82, DEGV01, Koc06]. Therefore, the data complexity of our higher-order query languages is in P.

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.

5.2 Complexity of evaluating HO

5.2.1 Complexity of evaluating lower degree terms

Degree 0 terms.
Naive evaluation of a degree 0 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 use Proposition 4.1 to reduce the problem to the evaluation for fragments of Datalog.

Proposition 5.1. *Evaluation of degree 0 terms over* RA *is linearly inter-reducible to NRDL$^\neg$ evaluation, while for* RA^+ *we can reduce to and from NRDL evaluation.*

This result gives us a PSPACE upper bound for the complexity of evaluating degree 0 terms over RA [Var82, Imm86, DEGV01].

The PSPACE upper bound is tight, even for terms with no negation and no union, and even for query complexity.

Proposition 5.2. *The evaluation problem for* $\mathsf{HO}_0^0[\mathsf{CQ}_C]$ *is* PSPACE*-hard.*

Proof. We reduce the *finite automaton intersection problem*, which is known to be PSPACE-hard from Lemma 3.2.3 in the work of Kozen [Koz77], to the evaluation problem for $\mathsf{HO}_0^0[\mathsf{CQ}_C]$. Formally, the finite automaton intersection problem consists of deciding, given m deterministic finite automata (DFA) $M_1, ..., M_m$, with a common alphabet Σ, whether an element of Σ^* is accepted by all $M_1, ..., M_m$.
Let $G_1, \ldots, G_m$ be the state diagrams for $M_1, \ldots, M_m$, respectively. For each $i \in [1, m]$, let u_i be the initial state of G_i. We can convert each DFA G_i to an equivalent one with only one final state as follows. We add a new final state v_i and add an ϵ-transition (a transition on the empty string) from each final state in G_i to v_i. We also add an ϵ-transition from v_i to v_i. Thus, without loss of generality, we assume each G_i has a single final state v_i, and there is a self-loop on v_i.

We denote by $N_1, ..., N_m$ the sets of states of $G_1, ..., G_m$, by $a_1, ..., a_m$, $b_1, ..., b_m$, $c_1, ..., c_m$, d some distinguished attribute names with domains $Dom(a_i) = Dom(b_i) =$

$Dom(c_i) = N_i$ and $Dom(d) = \Sigma$, by $\tau_1, ..., \tau_m$ the relational types (a_1, d, b_1), ..., (a_m, d, b_m), and by n the product $|N_1| \cdot ... \cdot |N_m|$. We identify each G_i with a relation of type τ_i that contains a tuple $t = (u, p, v)$ iff there is a transition from u to v through a p-labeled edge in G_i. We now introduce the following terms:

$$Q \;=\; \lambda R.\, \pi_A\big(\, \rho_{\{\bar{b}/\bar{c}\}}(R) \;\bowtie\; \rho_{\{\bar{a}/\bar{c}\}}(R) \,\big)$$

$$Q^n \;=\; \lambda R.\, \big(\underbrace{Q_m \circ ... \circ Q_m}_{\lfloor \lg(n) \rfloor + 1 \text{ times}}\big)\,(R)$$

where A is the set of attributes $\{a_1, ..., a_m, b_1, ..., b_m\}$, R is a relational variable of type $\bar{\tau} = (a_1, ..., a_m, b_1, ..., b_m)$, and Q, Q^n are CQ terms of type $\bar{\tau} \to \bar{\tau}$. The intuitive meaning of the above terms is as follows. The term Q receives as input (a relation representing) a state diagram $\bar{G}$ with states over $Dom(\tau_1) \times ... \times Dom(\tau_m)$ and it returns (the relation representing) the state diagram $\bar{G}^2$ that contains an edge $(\bar{u}', \bar{v}')$ whenever $\bar{G}$ contains the edges $(\bar{u}', \bar{z}')$ and $(\bar{z}', \bar{v}')$, for some intermediate vertex $\bar{z}'$. The term Q^n simply computes the $2^{\lfloor \lg(n) \rfloor + 1}$-fold iteration of the query Q on the input relation. Given the above definitions, it is clear that the following $\mathsf{HO}_0^0[\mathsf{CQ}_C]$ term

$$\Phi \;=\; \big(\pi_\emptyset \circ \sigma_{\substack{\bar{a}=\bar{u} \\ \bar{b}=\bar{v}}} \circ Q^n \circ \pi_A\big)\,(G_1 \bowtie ... \bowtie G_m)$$

evaluates to `true` iff each G_i contains a path p_i from u_i to v_i such that the label sequences on $p_1, \ldots, p_m$ are the same. Since there is a self-loop at v_i, the length of p_i can be assumed to be exactly $2^{\lfloor \lg(n) \rfloor + 1}$ ($\geq n \geq |G_1 \bowtie ... \bowtie G_m|$).

To conclude the proof, it is sufficient to observe that the term Φ above has size linear in the sum of the sizes of $G_1, ..., G_m$. This shows that the evaluation problem for relational terms of degree 0 is PSPACE-hard. □

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

Proposition 5.3. *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. □

Degree 1 Terms.

For degree 1 terms, which contain variables of order 1, we will make use of our

results from degree 0, plus observations inspired by Schubert's work [Sch01] on the complexity of normalization for low-degree terms in the standard λ-calculus.

The following result gives the complexity of degree 1 terms in the strongly-typed languages $\mathsf{HO}[\mathsf{RA}]$ and $\mathsf{HO}[\mathsf{RA}^+]$.

Theorem 5.4. *The problem of evaluating degree 1 terms over either* RA *or* RA^+ *is* EXPTIME*-complete.*

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

Proposition 5.5. *The problem of evaluating degree 1 terms over* RA *is in* EXPTIME*.*

Proof. We perform standard innermost-reduction to reduce a degree 1 term to a degree 0 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$^\neg$ expressions in exponential time. □

Proposition 5.6. *The problem of evaluating degree 1 terms is* EXPTIME*-hard, even for* $\mathsf{HO}[\mathsf{RA}^+]$.

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 ω with $|\omega| = n$. The DTM $\mathcal{M}$ is represented as a 5-tuple $(Q, \Sigma, \Gamma, \delta, q_0, \mathcal{F})$ with:

- Q: a finite set of states,
- Σ: the input alphabet: a finite set of symbols,
- $\Gamma \supseteq \Sigma \uplus \{\square\}$: the working tape alphabet: a finite set of symbols,
- $\delta : (Q \setminus F) \times \Gamma \to Q \times \Gamma \times \{01, 00, 10\}$: 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 Γ 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 δ is a partial function, we add $\delta(q, a) = (q, a, 00)$ for all (q, a) such that $\delta(q, a) = \emptyset$ to extend δ 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 δ, 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 $S(p_1, \ldots, p_n, a_1, \ldots, a_k, h, b_1, \ldots, b_m)$ (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 1 terms, presented in the proof of Proposition 4.2, is used to build a function τ of size $O(n)$ that takes a relation R of type $\mathcal{T}$ and a query Q of type $\mathcal{T}_Q = \mathcal{T} \to \mathcal{T}$, returning $Q^{2^n}(R)$.
- $T(\vec{b}, \vec{a}, \vec{b'}, \vec{a'}, c_1, c_2)$ is a relation that stores all the transitions of δ. 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 ($c_1c_2 = 01$), moves to the left ($c_1c_2 = 10$), or stays ($c_1c_2 = 00$).
- A relation $F(\vec{b})$ is used to store all the final states in $\mathcal{F}$.
- We use two degree 0 terms to represent Succ and Diff over any relation P of type $(p_1, \ldots, p_n)$. Both Succ and Diff are of type $(p_1, \ldots, p_n) \to (p_1, \ldots, p_n, v_1, \ldots, v_n)$, which is also denoted by $(\vec{p}) \to (\vec{p}, \vec{v})$.

  We define Diff as follows.

$$\mathsf{Diff} = \lambda P. \bigcup_{1 \le i \le n} (\sigma_{p_i=0, v_i=1}(P^*) \cup \sigma_{p_i=1, v_i=0}(P^*))$$

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

The following term defines Succ.

$$\mathsf{Succ} = \lambda P. \bigcup_{1 \le i \le n} \sigma_C \left(P \bowtie (\rho_{\vec{p}/\vec{v}}(P)) \right)$$

with

$$\begin{aligned} C \;=\; & (p_1 = v_1, \ldots, p_{i-1} = v_{i-1}, p_i = 0, v_i = 1, \\ & p_{i+1} = 1, \ldots, p_n = 1, v_{i+1} = 0, \ldots, v_n = 0) \end{aligned}$$

- Using Diff and Succ, we can define a term ρ_0 of type $(\vec{p}, \vec{a}, h, \vec{b}) \to (\vec{p}, \vec{a}, h, \vec{b})$ 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 ρ_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 transition.

Let S_0 be an instance of S that represents the input ω. 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 ω within 2^n steps. □

The following proposition states the query complexity of degree 1 terms.

Proposition 5.7. *The problem of evaluating degree 1 terms remains* EXPTIME*-hard when we fix the relational constants in the term.*

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

Now we can represent S_0 by the following expression:

$$S_0 = \bigcup_{1 \le i \le n} \left(\sigma_{\vec{A}=t_i} (\rho_{A/A_1}(\mathsf{D}_0) \bowtie \ldots \bowtie \rho_{A/A_m}(\mathsf{D}_0)) \right)$$

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

Adding recursion.
The language of *Inflationary Fixed Point logic* over a 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 heads, 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 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.

We now study the effect of the order 2 constant, `ifp`, on the evaluation problem of low degree terms. The proposition below shows that `ifp` evaluation can be reduced to evaluation with terms of degree 1 and only the standard relational query constants. It gives an alternative proof of Proposition 5.6, as explained below.

Proposition 5.8. *Evaluation of a degree 1 term in* $\mathsf{HO}[\mathsf{RA}]$ *or in* $\mathsf{HO}[\mathsf{RA}^+]$ *containing* `ifp` *is polynomially reducible to evaluation of a degree 1 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 $\texttt{ifp}(Q, \mathsf{D}_0)$ for any query Q and fixed D_0 of size at most n by $Q'^{2^n}(\mathsf{D}_0)$, where Q' is the modification of Q to union with its input. Moreover, there is a small degree 1 term that transforms any query Q of a given query type to Q' and there is a degree 1 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 5.6. Thus we can transform a term formed from `ifp` by replacing subterms $\texttt{ifp}(\rho, \rho')$ with a degree 1 term applied to ρ and ρ'. Doing this iteratively gives the desired transformation. □

The result above implies that the complexity of evaluating terms of degree from 1 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 0 evaluation.

Proposition 5.9. *Evaluation of a degree 0 term in* HO[RA$^+$] *or in* HO[RA] *containing* `ifp` *is* EXPTIME*-complete.*

The upper bound is inherited from the degree 1 case. Hardness follows from the EXPTIME-hardness of Datalog in query complexity [DEGV01], and the fact that Datalog can easily be embedded in `ifp`.

5.2.2 Complexity of evaluating arbitrary degree terms

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

Theorem 5.10. *The problem of evaluating degree k terms, with $k \geq 0$, is:*

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

Note: 0-EXPSPACE denotes PSPACE, and 0-EXPTIME denotes PTIME. As stated in Section 3.2, m-EXPTIME refers to the class of functions that run in time $\mathtt{exp}_m^{n^{O(1)}}$, and similarly for m-EXPSPACE.

When the terms are of degree 0, the evaluation problem is PSPACE-complete. When the degree of the terms is 1, the evaluation problem is EXPTIME-complete. The complexity of evaluation increases from m-EXPTIME (or m-EXPSPACE) to $(m+1)$-EXPTIME (or $(m+1)$-EXPSPACE) when the degree of the terms increases by 2. This is different from the upper bounds obtained by applying β-reduction, where the upper bounds exponentially increase when the degree of the terms increases by 1.

Upper bounds for evaluating degree k terms.
Based on the techniques in [HK96], we give the following upper bound.

Proposition 5.11. *The problem of evaluating degree k terms is:*

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

Proof. Two cases are proved separately:

- We show that the problem of evaluating degree k terms with $k = 2m - 1$ and $m \geq 1$ is in m-EXPTIME.

  We use m-hyperexponential time to reduce a degree $2m-1$ term τ to a degree $m-1$ term τ'. Similarly to the EXPTIME evaluation of degree 1 terms, we can evaluate τ' in m-hyperexponential time: simply reduce to degree 0, blowing up the size but not the arity, and then apply the evaluation strategy for degree 0. Thus, the problem is in m-EXPTIME.

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

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

□

The argument here relies on reduction to the degree 1 case via β-reduction, plus a few properties of the constants. The same argument is easily seen to hold for the extension with recursion.

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. Note that $\mathtt{exp}_m^n = 2^{2^{\cdot^{\cdot^{2^n}}}}$ with a tower of m 2's (see Section 3.2). Assuming X_0 is a HO term of type T^0 and Y_0 is a HO term of type $T^0 \to T^0$, we define *iteration functions* that return $Y_0^{\mathtt{exp}_{k+1}^n}$ when applying over X_0.

Proposition 5.12. *Given a* HO *term X_0 of type T^0 and of order k_0 and a* HO *term Y_0 of type $T^1 = T^0 \to T^0$, there exists a* HO^{k_0+k} *term that returns $Y^{\mathtt{exp}_k^n}(X)$.*

Proof. We first build the following term.

Subterm
$\tau_n^1 = \lambda x_{n-1}^1.\lambda x^0.x_{n-1}^1(x_{n-1}^1(x^0))\tau_{n-1}^1$
$\ldots$
$\tau_0^1 = \lambda x^0.Y(Y(x^0))$
$\ldots$
$\tau_n^i = \lambda x_{n-1}^i.\lambda x^{i-1}\ldots\lambda x^0.((x_{n-1}^i(x_{n-1}^i(x^{i-1})))\ldots x^0)\tau_{n-1}^i$
$\ldots$
$\tau_0^i = \lambda x^{i-1}\ldots\lambda x^0.(x^{i-1}(x^{i-1}(x^{i-2})))\ldots x^0$

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

$T^1 = T^0 \to T^0$

$T^2 = T^1 \to T^1 = T^1 \to T^0 \to T^0$

$T^i = T^{i-1} \to T^{i-1} = T^{i-1} \to \ldots \to T^0 \to T^0$

From these we define a term as follows:

$$\rho_n^k = ((\lambda Y.(\tau_n^k \tau_n^{k-1})\ldots\tau_n^1)Y_0)X_0$$

This term takes two terms X_0, Y_0 and returns $Y_0^{\mathtt{exp}_k^n}(X_0)$. □

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 5.13. *We can efficiently construct an order $m-2$ term Cell_{m-2} over RA^+ of some type Δ_{m-2} and an order $m-1$ term Succ_{m-2} of type $\Delta_{m-2} \to \Delta_{m-2}$ such that by iterating Succ_{m-2} on 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 Diff_{m-2} to compare the objects in $\mathcal{S}_{m-2}$. With respect to the semantics of Diff_{m-2}, $\mathcal{S}_{m-2}$ contains exp_m^n distinct objects.*

Proof. First, we explain the intuition of the proof.

- *Base case (m=2):* Let $\mathsf{D}_1, \ldots, \mathsf{D}_{n+1}$ be $n+1$ single attribute relational constants respectively of type $(a_1), \ldots, (a_n), b$. Each D_i contains two tuples $\langle 0 \rangle$ and $\langle 1 \rangle$,

i.e., $\mathsf{D}_i = \{\langle 0\rangle, \langle 1\rangle\}$ with $1 \leq i \leq n+1$. We build up an instance D_0 using the term $\mathsf{D}_1 \bowtie \ldots \bowtie \mathsf{D}_{n+1}$.

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 Cell_0 satisfies the last attributes of all the 2^n tuples equivalent to 0. We now describe Succ_0. Consider a number in 2^{2^n} as a sequence of 2^n bits. Succ_0 should find the first bit that is 0 and flip it to 1. Then we build $=_0$ and Diff_0.

- *Induction case*: We now show how to inductively define Cell_{k+1}, Succ_{k+1}, $=_{k+1}$, and Diff_{k+1} from Cell_k, Succ_k, $=_k$, and Diff_k. We define Cell_{k+1} as an order $k+1$ term of type $\Delta_k \to \mathcal{T}_0$ with $\mathcal{T}_0 = (u)$. Intuitively, Cell_{k+1} always returns $\{\langle 0\rangle\}$ when applying to an element of the set $\{\mathsf{Cell}_k, \mathsf{Succ}_k(\mathsf{Cell}_k), \ldots, (\mathsf{Succ}_k)^{\mathtt{exp}^n_{k+2}}(\mathsf{Cell}_k)\}$. Using iterators from Proposition 5.12, we define Succ_{k+1}, $=_{k+1}$, and Diff_{k+1}.

Below are the details of the proof.

Base case (m=2).
Let $\mathsf{D}_1, \ldots, \mathsf{D}_{n+1}$ be $n+1$ single attribute relational constants respectively of type $(a_1), \ldots, (a_n), b$. Each D_i contains two tuples $\langle 0\rangle$ and $\langle 1\rangle$, i.e. $\mathsf{D}_i = \{\langle 0\rangle, \langle 1\rangle\}$ with $1 \leq i \leq n+1$.

We build up an instance D_0 using the term $\mathsf{D}_1 \bowtie \ldots \bowtie \mathsf{D}_{n+1}$.

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}. Our function Succ will move from one such code of a number to the code of its successor.

Below we will often use $I_0 = \{\langle 0\rangle\}$ (resp., $I_1 = \{\langle 1\rangle\}$) to denote a relational constant of type (f) with one tuple $\{\langle 0\rangle\}$ (resp., $\{\langle 1\rangle\}$).

Towards building the initial position Cell_0, we build the following query Q_1 of type $(\vec{a}, b, \vec{p}, \vec{c}, b_0) \to (\vec{a}, b, \vec{p}, \vec{c}, b_0)$ with $\vec{c} = c_1, \ldots, c_n$.

$$\begin{aligned} Q_1 = \quad & \lambda S.\pi_{\vec{a},b}(S) \bowtie \pi_{\vec{v}}(\mathsf{Succ}(\pi_{\vec{p}}(S))) \bowtie \\ & (\pi_{\vec{c},b_0}(S) \cup \rho_{\vec{p}/\vec{c},b/b_0}(\pi_{\vec{p},b}(\sigma_{b=0}(\rho_{\vec{a}/\vec{p}}(S) \bowtie \pi_{\vec{p}}(S))))) \end{aligned}$$

with Succ built in Proposition 5.6.

Then we have $\mathsf{Cell}_0 = \pi_{\vec{c}}((Q_1)^{2^n}(R \bowtie D_0 \bowtie D_1))$ with D_0 of type $\vec{p}$, D_1 of type $\vec{c}$ and $D_0 = \{\langle 0, \ldots, 0, \rangle\}$, $D_1 = \{\}$.

We now describe Succ_0. Consider a number in 2^{2^n} as a sequence of 2^n bits. Succ_0 should find the first bit that is 0 and flip it to 1. It does this via iterating, using an additional bit to mark its place within the iteration.

We now give the formal definition.

Let Q_2 be a term of type $(\vec{a}, b, \vec{p}, \vec{c}, b_0, f) \to (\vec{a}, b, \vec{p}, \vec{c}, b_0, f)$ where f is for the flag bit.

$$Q_2 = \lambda S.\pi_{\vec{a},b}(S) \bowtie \rho_{\vec{v}/\vec{p}}(\pi_{\vec{v}}(\mathsf{Succ}(\pi_{\vec{p}}(S)))) \bowtie C$$

where C is the union of the following three terms:

$C_1 = (\pi_{\vec{c},b_0}(S) \cup \rho_{\vec{p}/\vec{c}}(\pi_{\vec{p}}(\sigma_{b=1,f=0}(S^*))) \bowtie \rho_{f/b}(I_0)) \bowtie I_0$
with $S^* = \rho_{\vec{a}/\vec{p}}(S) \bowtie \pi_{\vec{p}}(S)$.

$C_2 = (\pi_{\vec{c},b_0}(S) \cup \rho_{\vec{p}/\vec{c}}(\pi_{\vec{p}}(\sigma_{b=0,f=0}(S^*))) \bowtie \rho_{f/b}(I_1)) \bowtie I_1$

$C_3 = (\pi_{\vec{c},b_0}(S) \cup \rho_{\vec{p}/\vec{c},b/b_0}(\pi_{\vec{p},b}(\sigma_{f=1}(S^*)))) \bowtie I_1$

Using an iterator from Proposition 5.12, we define:

$$\mathsf{Succ}_0 = \lambda R^i.\pi_{\vec{c}}((Q_2)^{2^n}(R^i \bowtie D_0 \bowtie D_1))$$

Having defined Succ_0 we turn to defining equivalence between two codes of type Δ_0, which will be required inductively. First, let Q_3 be a term of type $(\vec{a}, b, \vec{c}, d, \vec{p}) \to (\vec{a}, b, \vec{c}, d, \vec{p})$.

$$Q_3(S_1, S_2, P) = C_4 \bowtie \pi_{\vec{a},b,\vec{c},d,\vec{v}}(S_1 \bowtie S_2 \bowtie \mathsf{Succ}(P))$$

with $C_4 = \sigma_{b=d}(\rho_{\vec{b}/\vec{p}}(S_1) \bowtie \rho_{\vec{c}/\vec{p}}(S_2) \bowtie P)$ checking if two bits in S_1 and S_2 at the current position P are equivalent.

We can now define the equivalence function as a boolean term over Δ_0 as follows.

$$S_1 =_0 S_2 \text{ iff } (Q_3)^{2^n}(S_1 \bowtie \rho_{\vec{a}/\vec{c},b/d}(S_2) \bowtie P_0) \neq \emptyset$$

with $P_0 = \{\langle 0, \ldots, 0\rangle\}$ an n attribute relational constant containing only one tuple.

We now define Diff_0 as follows.

$$\mathsf{Diff}_0(S_1, S_2) = \pi_{\emptyset}\left(\sigma_{\vec{a}=\vec{c},b\neq d}(S_1 \bowtie \rho_{\vec{a}/\vec{c},b/d}(S_2))\right)$$

with $b \neq d$ a short notation for the union of $(b = 0, d = 1)$ and $(b = 1, d = 0)$.

Induction case.

We now show how to inductively define Cell_{k+1}, Succ_{k+1}, $=_{k+1}$, and Diff_{k+1} from Cell_k, Succ_k, $=_k$, and Diff_k.

We define Cell_{k+1} as an order $k+1$ term of type $\Delta_k \to \mathcal{T}_0$ with $\mathcal{T}_0 = (u)$.

Intuitively, Cell_{k+1} always returns $\{\langle 0 \rangle\}$ when applying to an element of the set $\{\mathsf{Cell}_k, \mathsf{Succ}_k(\mathsf{Cell}_k), \ldots, (\mathsf{Succ}_k)^{\mathsf{exp}^n_{k+2}}(\mathsf{Cell}_k)\}$, which is denoted by $\{t_1, \ldots, t_N\}$ with $N = \mathsf{exp}^n_{k+2}$.

$$\mathsf{Cell}_{k+1} = \lambda x.\rho_{f/u}(I_0)$$

We note that the aggregates under $\bigcup$ and $\bigcap$ are easily built iterators defined in Proposition 5.12. For simplicity, we combine the condition and the object of a selection operator when the term is Boolean and simple. For example, we use $f(i) = a$ to represent $\pi_\emptyset(\sigma_{u=a}(f(i)))$.

Now, we define Succ_{k+1} to be the term below.

$$\lambda \mathbf{D}_{k+1}. \bigcup_{1 \leq i \leq N} \left(C(t_i) \bowtie (\mathbf{D}_{k+1}(t_i) = 0) \bowtie \bigwedge_{1 \leq j < i} (\mathbf{D}_{k+1}(t_j) = 1) \right)$$

with $C(t_i)$ the union of the following terms:

$$\begin{aligned} C_1(t_i) &= \textstyle\bigcup_{1 \leq l < i} \left(\lambda \mathbf{D}_k.(\mathbf{D}_k =_k t_l) \bowtie \rho_{f/u}(I_0)\right) \\ C_2(t_i) &= \lambda \mathbf{D}_k.(\mathbf{D}_k =_k t_i) \bowtie \rho_{f/u}(I_1) \\ C_2(t_i) &= \textstyle\bigcup_{i < l \leq N} \left(\lambda \mathbf{D}_k.(\mathbf{D}_k =_k t_l) \bowtie f(t_l)\right) \end{aligned}$$

The function $=_{k+1}$ is defined to be the following boolean term.

$$\lambda \mathbf{D}_{k+1}.\lambda \mathbf{D}'_{k+1}. \bigwedge_{\mathbf{D}_k \in \{t_1, \ldots, t_N\}} (\mathbf{D}_{k+1}(\mathbf{D}_k) = \mathbf{D}'_{k+1}(\mathbf{D}_k))$$

Similarly, the following term defines Diff_{k+1}.

$$\lambda \mathbf{D}_{k+1}.\lambda \mathbf{D}'_{k+1}. \bigvee_{\mathbf{D}_k \in \{t_1, \ldots, t_N\}} (\mathbf{D}_{k+1}(\mathbf{D}_k) \neq \mathbf{D}'_{k+1}(\mathbf{D}_k))$$

with $\mathbf{D}_{k+1}(\mathbf{D}_k) \neq \mathbf{D}'_{k+1}(\mathbf{D}_k)$ a short notation for $(\mathbf{D}_{k+1}(\mathbf{D}_k) = 0 \wedge \mathbf{D}'_{k+1}(\mathbf{D}_k) = 1) \vee (\mathbf{D}_{k+1}(\mathbf{D}_k) = 1 \wedge \mathbf{D}'_{k+1}(\mathbf{D}_k) = 0)$. □

Lower bounds for degree k terms.

Using Proposition 5.12, Proposition 5.13 and the techniques inspired by [HK96, Mai92], we can show that the preceding upper bounds are tight.

Proposition 5.14. *The problem of evaluating degree k terms is:*

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

Proof. We first give the intuition of the proof for both parts of the proposition.

- To show that the problem of evaluating degree k terms with $k = 2m - 1$ 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 $2m - 1$ to repeat that order m term m-hyperexponential times.
- To show that the problem of evaluating degree k terms with $k = 2m$ 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 $2m$ to repeat that order m term $(m + 1)$-hyperexponential times.

Details of the proof for both cases of k are given in more detail below.

For the case $k = 2m$, we give a reduction below from the acceptance problem of a DTM $\mathcal{M}$ running with time bound exp_k^n.

We use Cell_{m-2} and Succ_{m-2} from Proposition 5.13, such that the iteration of Succ_{m-2} on Cell_{m-2} generates a set of order $m - 2$ terms $\{t_1, \ldots, t_N\}$ of type Δ_{m-2} with $N = \mathsf{exp}_m^n$ which will represent indices for cells on the tape.

A tape configuration can then be represented by a term τ of type $\Delta_{m-2} \to (\vec{a}, \vec{b}, h)$ that maps each Δ_{m-2} object (in particular, each t_i) to the description of a cell. We will thus abbreviate $\Delta_{m-2} \to (\vec{a}, \vec{b}, h)$ by $\mathsf{TConfig}$ below, suppressing the dependence on m. Given an element t in Δ_{m-2}, we use $A(t)$, $B(t)$, and $H(t)$ to respectively denote $\pi_{\vec{a}}(\tau((t)))$, $\pi_{\vec{b}}(\tau((t)))$, and $\pi_h(\tau((t)))$. Intuitively, given a cell at the position represented by t, $A(t)$ denotes the symbol on that cell, $B(t)$ represents the state of the DTM, and $H(t)$ says whether the head is on that cell.

Similarly to the proof of Proposition 5.6, we use $T(\vec{b}, \vec{a}, \vec{b'}, \vec{a'}, c_1, c_2)$ to store transitions between states and head contents that are valid according to δ. T is an input-free term that builds an instance consisting of exactly the bit combinations $\vec{b}, \vec{a}, \vec{b'}, \vec{a'}, c_1, c_2$ that are valid.

Given an input ω of size n, we use a set of relational constants of type $(\vec{a})$: $R_1^\omega, \ldots, R_n^\omega$ to store its symbols. Each constant R_i^ω contains the binary form of the symbol at position i of ω. We also use one relational constant $R_1^\square$ to store the blank tape symbol. Similarly, we use a relational constant R^{q_0} of type $(\vec{b})$ to store the initial state. We use two single attribute relational constants U_0 and U_1 of type (h) such that $U_0 = \{\langle 0 \rangle\}$ and $U_1 = \{\langle 1 \rangle\}$.

The initial configuration of $\mathcal{M}$ is coded by the following order $(m-1)$ term of type $\mathsf{TConfig}$, named τ_0:

$$\tau_0 = \lambda\tau.(C_1 \cup C_2 \cup C_3)$$

with C_1, C_2, C_3 defined as follows.

$$C_1 = (\tau =_{m-2} \mathsf{Cell}_{m-2}) \bowtie R_1^\omega \bowtie R^{q_0} \bowtie U_1$$

$$\begin{aligned} C_2 = \bigcup_{1<i\leq n} \quad & (\tau =_{m-2} (\mathsf{Succ}_{m-2})^{i-1}(\mathsf{Cell}_{m-2})) \\ & \bowtie R_i^\omega \bowtie R^{q_0} \bowtie U_0 \end{aligned}$$

$$\begin{aligned} C_3 = \bigcup_{n<i\leq N} \quad & (\tau =_{m-2} (\mathsf{Succ}_{m-2})^{i-1}(\mathsf{Cell}_{m-2})) \\ & \bowtie R_i^\square \bowtie R^{q_0} \bowtie U_0 \end{aligned}$$

The running of the index i from n to N is easily coded using iteration functions.

We use an order m term ρ that transforms the current configuration, described by an element of type $\mathsf{TConfig}$, to the next configuration, also described by an element of type $\mathsf{TConfig}$.

We define ρ as below.

$$\begin{aligned} \rho = \lambda\tau.\lambda t. \bigcup_{1\leq i\leq N} \quad & (t =_{m-2} (\mathsf{Succ}_{m-2})^{i-1}(\mathsf{Cell}_{m-2})) \\ & \bowtie A'(t) \bowtie B'(t) \bowtie H'(t) \end{aligned}$$

Intuitively, when the current configuration is described by τ, we define $\rho(\tau) = \tau'$. We use $A'(t)$, $B'(t)$, and $H'(t)$ to respectively denote $\pi_{\vec{a}}(\tau'((t)))$, $\pi_{\vec{b}}(\tau'((t)))$, and $\pi_h(\tau'((t)))$.

The following term finds the position of the head and returns the required transition in T.

$$\Phi(t) = \pi_{\vec{b},\vec{a},\vec{b'},\vec{a'},c_1,c_2} (\sigma_{h=1}(H(t) \bowtie B(t) \bowtie A(t) \bowtie T))$$

This function will be called several times whenever we want to find the transition of the DTM.

The following term represents the change in the state.

$$B'(t) = \pi_{\vec{b'}} \bigcup_{1 \le i \le N} \Phi\left((\mathsf{Succ}_{m-2})^{i-1}(\mathsf{Cell}_{m-2})\right)$$

The following term represents the new symbol at a position defined by t:

$$A'(t) = \pi_{\vec{a'}}\left(\sigma_{h=1}(H(t) \bowtie \Phi(t))\right) \cup \pi_{\vec{a}}\left(\sigma_{h=0}(A(t) \bowtie H(t))\right)$$

For brevity, we use some abbreviations below:

Φ^* denotes $\Phi(s) \bowtie (s =_{m-2} (\mathsf{Succ}_{m-1})^{i-1}(\mathsf{Cell}_{m-2}))$

$next(s,t)$ denotes $(t =_{m-2} \mathsf{Succ}_{m-2}(s))$

$next(t,s)$ denotes $(s =_{m-2} \mathsf{Succ}_{m-2}(t))$

$\overline{next}(s,t)$ denotes $\mathsf{Diff}_{m-2}(t, \mathsf{Succ}_{m-2}(s))$

$\overline{next}(t,s)$ denotes $\mathsf{Diff}_{m-2}(s, \mathsf{Succ}_{m-2}(t))$

The following term defines the new position of the head.

$$\begin{aligned}
H'(t) = \ & \textstyle\bigcup_{1 \le i \le N} \pi_{\emptyset}(next(s,t) \bowtie (\sigma_{c_1c_2=01}(\Phi^*))) \bowtie U_1 \\
\cup \ & \textstyle\bigcup_{1 \le i \le N} \pi_{\emptyset}(next(t,s) \bowtie (\sigma_{c_1c_2=10}(\Phi^*))) \bowtie U_1 \\
\cup \ & \textstyle\bigcup_{1 \le i \le N} \pi_{\emptyset}((t =_{m-2} s) \bowtie (\sigma_{c_1c_2=00}(\Phi^*))) \bowtie U_1 \\
\cup \ & \textstyle\bigcup_{1 \le i \le N} \pi_{\emptyset}(\overline{next}(s,t) \bowtie (\sigma_{c_1c_2=01}(\Phi^*))) \bowtie H(t) \\
\cup \ & \textstyle\bigcup_{1 \le i \le N} \pi_{\emptyset}(\overline{next}(t,s) \bowtie (\sigma_{c_1c_2=10}(\Phi^*))) \bowtie H(t)
\end{aligned}$$

We now use our iterators from Proposition 5.12 to iterate through time rather than space. That is, we define a degree $2m-1$ term ρ^* that iterates the order m term ρ $\mathtt{exp}_m^n$ times over τ_0, i.e. $\rho^* = \rho^{\mathtt{exp}_m^n}(\tau_0)$. Now we have $\mathcal{M}$ accepts ω after $\mathtt{exp}_m^n$ steps if the state is in $\mathcal{F}$, i.e. $(\rho^*(t_1) \bowtie F) \neq \emptyset$.

For the case $k = 2m$, we give a reduction which is similar to that of the case $k = 2m-1$ except in the last step we define a degree $2m$ term ρ^{**} instead of ρ^*. This degree $2m$ term iterates the order m term ρ $\mathtt{exp}_{m+1}^n$ times over τ_0, i.e. $\rho^{**} = \rho^{\mathtt{exp}_{m+1}^n}(\tau_0)$. It is possible to decide the acceptance of a m-EXPSPACE DTM after $\mathtt{exp}_{m+1}^n$ steps. Thus we have $\mathcal{M}$ accepts ω within $\mathtt{exp}_m^n$ space if the state is in $\mathcal{F}$, i.e. $(\rho^{**}(t_1) \bowtie F) \neq \emptyset$. □

Similarly to Proposition 5.7, we can show that the complexity of evaluation does not change in some particular cases.

Proposition 5.15. *When we fix either the relational constants or the arities of all relational types occurring in the term the hardness results in Proposition 5.14 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 5.16. *The evaluation problem for higher-order terms has non-elementary complexity.*

5.2.3 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.

Definition 7. *A closed term τ is* linear *iff τ does not contain two occurrences of the same variable.*

The following result is clear:

Theorem 5.17. *A linear term $\tau \in \mathsf{HO}[\mathcal{F}]$ can be transformed in linear time to an equivalent Relational Algebra expression over $\mathcal{F}$, using standard β-reduction.*

Proof. We can reduce τ to an equivalent algebra expression τ' over $\mathcal{F}$ tractably via substitution. By the definition of linear terms, the size of τ' is the same as the size of τ. □

Thus, evaluating a linear term $\tau \in \mathsf{HO}[\mathcal{F}]$ is polynomially reducible to evaluating an expression in $\mathcal{F}$, which is NP-complete for 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.

Definition 8. *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 each variable Q in the term never occurs as an argument of Q. For example, $\lambda Q.\lambda D.Q(Q(D)$ has a self-nested variable, but $\lambda Q.\lambda D_1.\lambda D_2.Q(D_1) \bowtie Q(D_2)$ 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 0 terms using only polynomial space.

Proposition 5.18. *There is a* PSPACE *algorithm that reduces a degree k term τ^k of size n to a degree 0 term τ^0. The height of the construction tree of τ^0 is bounded by $O(n)$.*

Proof. Given a degree k term τ^k and its construction tree ξ, we go top-down through ξ 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 ≥ 1, then its left child must be of the form λX 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 ≥ 1,
  then return **Reduction**($root(\mathcal{L}(X)), \mathcal{L}(X)$).

- If T does not have variables of order ≥ 1,
  then return T.

The initial call is **Reduction**($root(\xi), \xi$). The output of **Reduction**($root(\xi), \xi$) is the construction tree of an equivalent term τ^0 without variables of order more than

0. 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 $O(n)$. □

From the proposition above, we can use a PSPACE algorithm to reduce an un-nested term of size n to an NRDL$^\neg$ query `Q`. The rank of the predicates in `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 NRDL$^\neg$ query can be evaluated using $O(n)$ space.

Proposition 5.19. *There is an evaluation algorithm for 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 NRDL$^\neg$ queries.

The two propositions above directly imply the following result.

Theorem 5.20. *The evaluation problem for un-nested terms in* HO[RA] *is in* PSPACE.

The same argument reducing to the degree 0 case can be used for `ifp`, yielding the following:

Proposition 5.21. *The evaluation problem for un-nested terms in* HO[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 0 terms, it is sufficient to fix only the arities of variables.

Proposition 5.22. *When we fix the arities of the relational variables, the evaluation problem for degree 0 terms is* NP*-complete for* RA$^+$ *and* RA.

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

However, to reduce the complexity of terms of degree higher than 0, we need to fix both the relational constants and the size of the variables.

Proposition 5.23. *When we fix all the relational constants and the size of all the variables, the evaluation problem for degree 1 terms is* NP*-complete for* RA^+*, and* PSPACE*-complete for* RA*.*

Proof. NP-hardness is obvious from Proposition 5.22, 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 $\mathcal{T} \restriction D$ be the restriction of 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. $(\mathcal{T} \to \mathcal{T}') \restriction D$ is $(\mathcal{T} \restriction D) \to (\mathcal{T}' \restriction D)$. Since D is finite, each $\mathcal{T} \restriction D$ 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 τ of type $\mathcal{T}$ we define its restriction $\tau \restriction D$, which will be of type $\mathcal{T} \restriction D$.

An *extension guess* for order 0 term τ is a mapping taking every variable of τ of type T to an object of type $T \restriction D$. A guess g is extended to all subterms of τ 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 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 τ 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 τ_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 5.24. *When we fix all the relational constants and the size of all the variables, the evaluation problem is* NP*-complete for* RA^+*, and in* PSPACE *for* RA *(hence* PSPACE*-complete for* RA *when the degree is at least 1).*

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 NRDL, NRDL$^\neg$, `ifp`, respectively). Hence:

Proposition 5.25. *The evaluation problem for fixed terms of query type for any of our signatures is in P.*

5.3 Complexity of evaluating HOCV and XQH

This section studies the complexity of evaluating HOCV and XQH of arbitrary degree. We first look at the complexity of evaluation for HOCV^0 terms, i.e. terms with only abstraction over nested relational variables. The equivalence results in the Subsection 4.2.3 and Koch's complexity results in [Koc06] give bounds on their evaluation:

Proposition 5.26. *The evaluation problem for* HOCV^0 *with* $[=_{atomic}]$ *without negation (resp., with negation) is* NEXPTIME*-complete (resp., TA*$[2^{O(n)}, O(n)]$*-complete).*

The evaluation problem for HOCV^0 *with* $[=_{deep}]$ *is TA*$[2^{O(n)}, O(n)]$*-hard and in* EXPSPACE*.*

Note that there is still a gap in the complexity of evaluating HOCV^0 with $[=_{deep}]$, which is TA$[2^{O(n)}, O(n)]$-hard and in EXPSPACE.

Before turning to terms that may include variables of order higher than 0, we state a result about β-reduction that will be useful.

Proposition 5.27. *When reducing from a* HOCV^k *term to an* HOCV^{k-1} *term with* $k \geq 1$*, the size of basic subterms (terms whose parse tree does not contain @ nodes or* λ *nodes) does not increase.*

Proof. We give an inner-most β-reduction algorithm **ReduceVar**, which takes as input a construction tree ξ and returns ξ' without order k variables.

- Let r_0 be the root of ξ, ξ_L and ξ_R respectively are its left and right subtree. If r_0 is @ and its left-child is λX, then:
  - If ξ_R does not have order k variables, then return the tree underneath λX where all X are substituted by ξ_R.

  - Else, return the tree with root r_0 where the left subtree is ξ_L and the right subtree is replaced by $\mathbf{ReduceVar}(\xi_R)$.

- Else if r_0 is not a leaf, return the tree with root r_0 where the left subtree is replaced by $\mathbf{ReduceVar}(\xi_L)$ and the right subtree is replaced by $\mathbf{ReduceVar}(\xi_R)$.

During the reduction, each order k variable X^k is substituted by an order k subtree ξ^k. After the substitution, the parent node of ξ^k is not a query constant node because ξ^k is of order k. The parent node is also not a variable node because variable nodes are leaves. Thus the parent node of ξ^k is either an @ node or a λ node. This means that we do not combine ξ^k with other basic subtrees to produce a bigger basic tree. □

The following theorem shows that the evaluation problem for terms with query variables remains in EXPSPACE. Thus the currently known worst-case bound is no worse when query variables are added. Furthermore, we show that the upper bound is now tight.

Theorem 5.28. *The evaluation problem of* HOCV^1 *with either* $[=_{atomic}]$ *or* $[=_{deep}]$ *is* EXPSPACE*-complete.*

Proof. Membership: Given an HOCV^1 term τ, we build its construction tree ξ and perform standard innermost-reduction to reduce ξ to an HOCV^0 construction tree ξ'. In the process, the size of the term increases exponentially.

Due to Proposition 5.27, the size of basic trees in ξ' is linear in the size of basic trees in ξ.

Evaluating ξ' is polynomially reducible to evaluating a set of HOCV^0 basic trees. The number of basic trees is polynomial in the size of τ', i.e. exponential in the size of τ, but the size of each tree is linear in the size of τ. A top-down algorithm is employed to evaluate each basic tree. Since the evaluation problem for HOCV^0 is in EXPSPACE [Koc06], we can evaluate each basic tree in exponential space. Since the number of rules is exponential, we can evaluate all of them using exponential space.

Hardness: We give a reduction from the acceptance problem of an exponential space deterministic Turing machine to this problem. We assume that the input of the Turing machine is ω and of length n. We will adapt the proof for the NEXPTIME-hardness in [Koc06] to code configurations of exponential size using HOCV^0 terms. In Koch's work, a composition operator is used, whereas, here we use a term $\mathcal{T} = t_n$ containing order 0 variables to repeat the product operator an exponential number of

times on a tuple $D_0 = \langle S : \Sigma, H : \langle 0, 1\rangle\rangle$, which contains the alphabet and the head position:

$$\begin{aligned} t_n &:= (\lambda R_{n-1}.(R_{n-1} \times R_{n-1}))\tau_{n-1} \\ &\dots \\ t_0 &:= D_0 \times D_0 \end{aligned}$$

where $\times$ is defined as follows:

$$u \times v := flatmap(pairwith_r)(pairwith_l(\langle l : u, r : v\rangle))$$

Notice that $\mathcal{T}$ contains all the lists of exponential size, each of which represents the positions and values of an exponential number of cells.

We define $\mathcal{C} = \mathcal{T} \times \mathcal{Q}$ where $\mathcal{Q}$ contains information about the state of TM.

The equivalence operator equ (also denoted as $=_{deep}$) between two nested relations is defined over the atomic equivalence operator $[=_{atomic}]$. For clarity, the λ terms are represented in the similar form of the "let" operator.

$A =_{deep} B = \mathsf{equ}(A, B) := eq^{n-1}(\pi_l(A), \pi_l(B)) \cap eq^{n-1}(\pi_r(A), \pi_r(A))$

...

$\mathsf{equ}^k(A, B) := eq^{k-1}(\pi_l(A), \pi_l(B)) \cap eq^{k-1}(\pi_r(A), \pi_r(A))$

$\mathsf{equ}^0(A, B) := A =_{atomic} B$

Assuming $n = 2^k$, the initial configuration of the TM is coded by *checkInit*, which is conjunction of the following terms:

$\pi_l(\dots(\pi_l)\mathcal{C}) = \omega$ where the number of π_l is $n - k$.

$\pi_r(\mathcal{C}) = Blank^{n-1} \cap \dots \pi_r(\dots \pi_r(\mathcal{C})) = Blank^k$ where the number of π_r is $n - k$ and $Blank^i$ represents a nested relation with height i and all leaves $Blank$.

The final configuration is checked by the following term: $checkFinal := \sigma_Q \in \mathcal{F}(\pi_Q(\mathcal{C}))$.

We also adapt the techniques of the NEXPTIME-hardness proof in [Koc06] to construct a HOCV^0 term ϕ_{succ} that determines, given a pairs of configurations, whether the first one can reach the other in one transition of the machine.

The idea of ϕ_{succ} is that it uses composition and binary search to compare two tuples of exponential size.

We define $\phi_{succ} = Next_n$ a query over a relation R of type $\langle A : \mathcal{T}_A, Q_1 : \mathcal{T}_{Q_1}, B : \mathcal{T}_B, Q_2 : \mathcal{T}_{Q_2}\rangle$. Intuitively, $Next_n$ returns all pair of configurations such that the first can reach the second. The subscript n denotes the height of the tree tuples used to represent the configurations. $Next_n$ is inductively defined as a union of the following queries:

$\mathsf{equ}(\pi_l(\pi_A(R)), \pi_l(\pi_B(R))) \cap Next_{n-1}(\pi_r(\pi_A(R)), \pi_r(\pi_B(R)))$
$\mathsf{equ}(\pi_r(\pi_A(R)), \pi_r(\pi_B(R))) \cap Next_{n-1}(\pi_l(\pi_A(R)), \pi_l(\pi_B(R)))$
$Side_n(\pi_l(\pi_A(R)), \pi_r(\pi_A(R)), \pi_l(\pi_B(R)), \pi_r(\pi_B(R)))$
$Next_0(\pi_l(\pi_A(R)), \pi_r(\pi_A(R)), \pi_l(\pi_B(R)), \pi_r(\pi_B(R)))$

where $Side_n(X_1, X_2, Y_1, Y_2)$ is inductively defined as union of $Side_{n-1}$ and $Side_0$:
$Side_{n-1}(\pi_r(X_1), \pi_l(X_2), \pi_r(Y_1), \pi_l(Y_2)) \cap \mathsf{equ}(\pi_l(X_1), \pi_l(Y_1)) \cap \mathsf{equ}(\pi_r(X_2), \pi_r(Y_2))$

where $Next_0(X_1, X_2, Y_1, Y_2)$ and $Side_0(X_1, X_2, Y_1, Y_2)$ are defined based on the transition function δ of the TM. For example:
$\pi_S X_1 =_{atomic} a \cap \pi_H X_1 =_{atomic} 1 \cap \pi_S Y_1 =_{atomic} b \cap \pi_H Y_2 =_{atomic} 1 \cap \pi_Q(R) =_{atomic} q_1 \cap \pi_Q(R) =_{atomic} q_2 \cap \delta(q_1, a, right) = (q_2, b)$

Note that the equality between an expression and an atomic value above can be easily coded in HOCV.

Using variables of order 0 and 1, we now build an HOCV^1 term $\tau = \tau_n$ that iterates ϕ_{succ} a doubly-exponential number of times on a relation R. First, the following terms are defined:

$$\begin{aligned} \tau_n &:= (\lambda Q_1.\lambda R_1.(Q_1(Q_1(R_1))))\tau_{n-1} \\ &\dots \\ \tau_0 &:= \lambda R.(\phi_{succ}(\phi_{succ}(R))) \end{aligned}$$

where $R = checkInit(\pi_l(\mathcal{C} \times \mathcal{C}))$ Then, we use $checkFinal(\pi_r(\tau))$ to check if there exists a doubly-exponential path from the initial configuration to a final one. Checking the acceptance of an exponential space DTM requires at most doubly-exponential steps. Thus τ returns a non-empty output if and only if the DTM accepts its input. □

For terms containing variables of order 2, the complexity goes up significantly.

Note that we require exponential space to evaluate deep equality using atomic equality. The EXPSPACE-hardness complexity of evaluating terms containing variables of order 1 implies that given $k \geq 1$, the problem of evaluating HOCV^k with deep equality is in the same complexity class with the problem of evaluating HOCV^k with atomic equality.

Theorem 5.29. *The evaluation problem of* HOCV^2 *with either* $[=_{atomic}]$ *or* $[=_{deep}]$ *is 2-*EXPTIME*-complete.*

Proof. Membership: We use doubly-exponential time to reduce an input HOCV^2 term τ to an equivalent HOCV^0 term τ'. Together with the results in Proposition 5.27, we have that the construction tree ξ' of τ' contains 2^{2^n} basic subtrees of linear size in the size of τ. Using bottom-up evaluation, we can evaluate each basic tree in 2^{2^m} time. Thus we can evaluate ξ' in doubly-exponential time.

Hardness: The hardness is shown in a similar way to the hardness proof of Theorem 5.28. A reduction from the acceptance problem of a doubly-exponential DTM to this problem is given. We use a nested relational instance to represent a configuration of the DTM. Then a query Q is used to generate next valid configurations. By repeating the query Q 2^{2^n} times, we can check the acceptance of a DTM within 2^{2^n} steps. The higher-order term that is used to repeat a query a doubly-exponential number of times is defined as follows.

$$\begin{aligned}
\tau_n^1 &:= (\lambda Q.\lambda R.(Q(Q(R))))\tau_{n-1} \\
&\dots \\
\tau_0^1 &:= \lambda R.Y(Y(R)) \\
\tau_n^2 &= \lambda x_{n-1}^2.\lambda Q.\lambda R.((x_{n-1}^2(x_{n-1}^2(Q)))R)\tau_{n-1}^2 \\
&\dots \\
\tau_0^2 &= \lambda Q.\lambda R.Q(R)
\end{aligned}$$

where all τ_i^1 are of type $\mathcal{T}^1 = T^0 \to T^0$ and all τ_i^2 are of type $T^2 = T^1 \to T^1 = T^1 \to T^0 \to T^0$

From that we define the following term that repeats Y_0 doubly-exponentially many times on X_0.

$$\rho_n^2 = (\lambda Y.(\tau_n^2\tau_n^1)Y_0)X_0$$

□

In general, we can ascertain the complexity of evaluating HOCV terms of arbitrary degree.

Theorem 5.30. *Given $k \geq 1$, the problem of evaluating HOCV^k with either $[=_{atomic}]$ or $[=_{deep}]$ is:*

- *m-$\mathsf{EXPSPACE}$-complete if $k = 2m - 1$, i.e. k is odd,*
- *$(m+1)$-$\mathsf{EXPTIME}$-complete if $k = 2m$, i.e. k is even.*

Proof. Intuitively, the membership is shown by first reducing variables of higher order using β-reduction, then evaluating the term containing variables of lower order. The hardness is shown in a similar way as the proof of Proposition 5.14 in Subsection 5.2.2, which shows similar bounds in the relational case, but roughly one exponential lower than the ones here. The main difference is that a nested relational type can represent a set of doubly-exponential cardinality, whereas a relational type can only represent a set of exponential cardinality.

Membership.

- $k = 2m - 1$: We reduce an HOCV^{2m-1} term τ to an equivalent HOCV^{m-1} term τ'. There are only order $(m-1)$ variables in τ'. Thus we can guess them using m-EXPSPACE. Having guessed a term,we can evaluate it in m-EXPSPACE.
- $k = 2m$: We reduce an HOCV^{2m} term τ to an equivalent HOCV^{m-1} term τ'. We use a bottom-up algorithm to evaluate τ'. Since there are only order $(m-1)$ variables in τ', we can evaluate each variable in $(m+1)$-EXPTIME. The size of τ' is $(m+1)$-hyperexponential in the size of τ. Thus we can evaluate τ' in $(m+1)$-EXPTIME.

Hardness.
Coding: Before showing the lower bound, we use the result from Proposition 5.12 to obtain the following result for HOCV.

Proposition 5.31. *Given a* HOCV *term* X_0 *of type* T^0 *and of order* k_0 *and a* HOCV *term* Y *of type* $T^1 = T^0 \to T^0$*, there exists a* HOCV^{k_0+k} *term that returns* $Y^{\mathtt{exp}^n_k}(X)$.

As in Proposition 5.13, we use these iterators to capture both a large amount of space and a large amount of time.

Proposition 5.32. *We can efficiently construct an order* $(m-2)$ HOCV *term* Cell_{m-2} *of some type* Δ_{m-2} *and an order* $(m-1)$ HOCV *term* Succ_{m-2} *of type* $\Delta_{m-2} \to \Delta_{m-2}$ *such that by iterating* Succ_{m-2} *on* 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* Diff_{m-2} *to compare the objects in* $\mathcal{S}_{m-2}$*. With respect to the semantics of* Diff_{m-2}*,* $\mathcal{S}_{m-2}$ *contains* exp^n_{m+1} *distinct objects.*

We omit the proof of this proposition, which is similar to Proposition 5.13 for higher-order relational queries. In the relational higher-order setting we can get exp^n_m distinct objects via iterating such a term, while the proposition above claims that

in our setting we can achieve exp^n_{m+1} distinct objects. This exponential increase is due to the fact that order 0 HOCV terms can code doubly-exponentially many states, whereas Relational Algebra terms can output objects of size at most exponential in the input, and hence can code only exponentially many states.

Reduction from the acceptance problem of a DTM.
We consider the case where k is even in more detail. For the case where k is odd, the proof is similar. Assuming $k = 2m$, we give a reduction below from the acceptance problem of a DTM $\mathcal{M}$ running with time bound exp^n_{m+1}.

We use Cell_{m-2} and Succ_{m-2} from Proposition 5.13, such that the iteration of Succ_{m-2} on Cell_{m-2} generates a set of order $m-2$ terms $\{t_1, \ldots, t_N\}$ of type Δ_{m-2} with $N = \mathsf{exp}^n_{m+1}$ which will represent indices for cells on the tape.

A tape configuration can then be represented by a term τ of type $\Delta_{m-2} \to \langle A : \tau_A, B : \tau_B, H : \tau_H \rangle$ that maps each Δ_{m-2} object (in particular, each t_i) to the description of a cell. We will thus abbreviate $\Delta_{m-2} \to \langle A : \tau_A, B : \tau_B, H : \tau_H \rangle$ by $\mathsf{TConfig}$ below, suppressing the dependence on m. Given an element t in Δ_{m-2}, we use $A(t)$, $B(t)$, and $H(t)$ to respectively denote $\pi_A(\tau((t)))$, $\pi_B(\tau((t)))$, and $\pi_H(\tau((t)))$. Intuitively, given a cell at the position represented by t, $A(t)$ denotes the symbol on that cell, $B(t)$ represents the state of the DTM, and $H(t)$ says whether the head is on that cell.

We use an input-free term T of type $\langle B : \tau_B, A : \tau_A, B' : \tau_B, A' : \tau_A, C : \tau_C \rangle$ to store transitions between states and head contents that are valid. In addition to the representation of the cells, we use C to denote the head's moving direction, which is either $left, right$ or $stand$.

Given an input ω of size n, we use a set of relations of type $\langle A : \tau_A \rangle$: $R^\omega_1, \ldots, R^\omega_n$ to store its symbol. Each relation R^ω_i contains the binary form of the symbol at position i of ω. We also use $R^\square_1$ to store the blank node symbol. Similarly, we use a relational constant R^{q_0} of type $\langle B : \tau_B \rangle$ to store the initial state. We also use two tuples $U_0 = \langle 0 \rangle$ and $U_1 = \langle 1 \rangle$ of type $\langle H : \tau_H \rangle$. The initial configuration of $\mathcal{M}$ is coded by the following order $(m-1)$ term of type $\mathsf{TConfig}$, named τ_0.

$$\tau_0 = \lambda\tau.(C_1 \cup C_2 \cup C_3)$$

with C_1, C_2, C_3 defined as follows.

$$C_1 = (\tau =_{m-2} \mathsf{Cell}_{m-2}) \times R^\omega_1 \times R^{q_0} \bowtie U_1$$

$$C_2 = \bigcup_{1<i\leq n} (\tau =_{m-2} (\mathsf{Succ}_{m-2})^{i-1}(\mathsf{Cell}_{m-2})) \times R^\omega_i \times R^{q_0} \times U_0$$

$$C_3 = \bigcup_{n<i\leq N}(\tau =_{m-2} (\mathsf{Succ}_{m-2})^{i-1}(\mathsf{Cell}_{m-2})) \times R_i^{\square} \times R^{q_0} \times U_0$$

The running of the index i from n to N is easily coded using iteration functions. The $\times$ operator and also the $\bowtie$ operator are readily coded by HOCV query constants, i.e. Monad Algebra operators [Koc06].

We use an order m term ρ that transforms the current configuration, described by an element of type $\mathsf{TConfig}$, to the next configuration, also described by an element of type $\mathsf{TConfig}$.

We define ρ as below.

$$\rho = \lambda\tau.\lambda t. \bigcup_{1\leq i\leq N} (t =_{m-2} (\mathsf{Succ}_{m-2})^{i-1}(\mathsf{Cell}_{m-2})) \times A'(t) \times B'(t) \times H'(t)$$

Intuitively, when the current configuration is described by τ, we define $\rho(\tau) = \tau'$. We use $A'(t)$, $B'(t)$, and $H'(t)$ to respectively denote $\pi_A(\tau'((t)))$, $\pi_B(\tau'((t)))$, and $\pi_H(\tau'((t)))$.

The following term finds the position of the head and returns the required transition in T.

$$\Phi(t) = \pi_{B,A,B',A',H}\left(\sigma_{H=1}(H(t) \bowtie B(t) \bowtie A(t) \bowtie T)\right)$$

This function will be called several times whenever we want to find the transition of the DTM.

The following term represents the change in the state.

$$B'(t) = \pi_B \bigcup_{1\leq i\leq N} \Phi\left((\mathsf{Succ}_{m-2})^{i-1}(\mathsf{Cell}_{m-2})\right)$$

The following term represents the new symbol at a position defined by t:

$$A'(t) = \pi_A\left(\sigma_{H=1}(H(t) \bowtie \Phi(t))\right) \cup \pi_A\left(\sigma_{H=0}(A(t) \bowtie H(t))\right)$$

For brevity, we use some abbreviations below:

Φ^* denotes $\Phi(s) \bowtie (s =_{m-2} (\mathsf{Succ}_{m-1})^{i-1}(\mathsf{Cell}_{m-2}))$

$next(s,t)$ denotes $(t =_{m-2} \mathsf{Succ}_{m-2}(s))$

$next(t,s)$ denotes $(s =_{m-2} \mathsf{Succ}_{m-2}(t))$

$\overline{next}(s,t)$ denotes $\mathsf{Diff}_{m-2}(t, \mathsf{Succ}_{m-2}(s))$

$\overline{next}(t,s)$ denotes $\mathsf{Diff}_{m-2}(s, \mathsf{Succ}_{m-2}(t))$

The following term defines the new position of the head.

$$\begin{aligned} H'(t) = \; & \bigcup_{1\le i\le N} \pi_{\emptyset}(next(s,t) \bowtie (\sigma_{C=right}(\Phi^*))) \bowtie U_1 \\ \cup \; & \bigcup_{1\le i\le N} \pi_{\emptyset}(next(t,s) \bowtie (\sigma_{C=left}(\Phi^*))) \bowtie U_1 \\ \cup \; & \bigcup_{1\le i\le N} \pi_{\emptyset}((t =_{m-2} s) \bowtie (\sigma_{C=stand}(\Phi^*))) \bowtie U_1 \\ \cup \; & \bigcup_{1\le i\le N} \pi_{\emptyset}(\overline{next}(s,t) \bowtie (\sigma_{C=right}(\Phi^*))) \bowtie H(t) \\ \cup \; & \bigcup_{1\le i\le N} \pi_{\emptyset}(\overline{next}(t,s) \bowtie (\sigma_{C=left}(\Phi^*))) \bowtie H(t) \end{aligned}$$

We now use our iterators from Proposition 5.31 to iterate through time rather than space. That is, we define a HOCV^{2m} term ρ^* that iterates the degree m term $\rho\ \mathtt{exp}^n_{m+1}$ times over τ_0, i.e. $\rho^* = \rho^{\mathtt{exp}^n_{m+1}}(\tau_0)$. Now we have $\mathcal{M}$ accepts ω after $\mathtt{exp}^n_{m+1}$ steps if the state is in $\mathcal{F}$, i.e. $(\rho^*(t_1) \bowtie \mathcal{F}) \neq \emptyset$ with $\mathcal{F}$ the set of final states. □

Lowering the complexity.
The complexity of evaluating higher-order complex-valued languages is related to two factors; the complexity of λ-reduction, and the complexity of lower-order evaluation. In the case of XML and complex values, the complexity of lower-order evaluation is in turn related to the ability to create and iterate over large intermediate structures. We consider how to eliminate the first factor, using the same restriction as in the relational case (see Subsection 5.2.3).

A variable $x \in \tau$ is *self-nested* if x occurs in two subtrees s, t of the construction tree of τ and two roots of s and t are linked to the same @ node.

Similar to the case of HO, an HOCV term is un-nested if a variable never occurs as an argument of itself in the term.

Definition 9. *A term is self-nested free (or un-nested) if it does not contain any self-nested variable.*

The complexity of evaluating self-nested free terms is much lower than the complexity of evaluating normal terms, matching the best known upper bound for ordinary Core XQuery.

Theorem 5.33. *The evaluation problem for un-nested* HOCV^n *terms with* $n \geq 1$ *is* EXPSPACE*-complete.*

Theorem 5.33 follows from a more general result, that allows the nesting restriction to hold only above some order.

Theorem 5.34. *Let* $\mathsf{HOCV}^n[m]$ *with* $1 \leq m \leq n$ *be the set of* HOCV^n *terms where all variables of order higher than* m *are un-nested. The evaluation problem for* $\mathsf{HOCV}^n[m]$ *is* $(m+1)/2$-$\mathsf{EXPSPACE}$*-complete if* m *is odd or* $(m/2+1)$-$\mathsf{EXPTIME}$*-complete if* m *is even, which are the bounds for* HOCV^m*.*

Proof. We show the theorem by giving a reduction from the evaluation problem for $\mathsf{HOCV}^n[m]$ terms to the evaluation problem for HOCV^m terms. Given an HOCV^n term τ^n and its construction tree ξ, we proceed top-down through ξ to find pairs consisting of an order n 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 an 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 $k > m$, then its left child must be of the form λX 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 n, then return **Reduction**($root(\mathcal{L}(X)), \mathcal{L}(X)$).

- If T does not have variables of order n, then return T.

The initial call is **Reduction**($root(\xi), \xi$). The output of **Reduction**($root(\xi), \xi$) is the construction tree of an equivalent term τ^m without variables of order greater than m. An implementation needs to explore only one branch at a time. The output of one branch can be guessed using exponential space, which is no more than the space required for evaluating HOCV^m terms.

In addition, 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, which is the length of one branch, is bounded by $O(n)$. This implies that we can evaluate the reduced term τ^m in $(m+1)/2$-$\mathsf{EXPSPACE}$ if m is odd or in $(m/2+1)$-$\mathsf{EXPTIME}$ if m is even. □

5.4 Conclusions

Table 5.1 summarizes the main complexity results for the higher-order language over relational databases. We have shown that the evaluation problem has non-elementary complexity, as one might expect from prior results in the λ-calculus. Restrictions on nesting lead to drastic reductions. 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.

Degree	Signature	General	Un-nested
0	RA,RA$^+$	PSPACE	undefined
	RA+ifp	EXPTIME	undefined
$2m$	RA,RA$^+$	m-EXPSPACE	PSPACE
	RA+ifp	m-EXPSPACE	EXPTIME
$2m-1$	RA,RA$^+$	m-EXPTIME	PSPACE
	RA+ifp	m-EXPTIME	EXPTIME

Table 5.1: Complexity of evaluation for HO.

Table 5.2 summarizes the polynomial reducibility between the XQuery languages and fragments of the higher-order language, and the complexity of the evaluation problem for those languages. In the table, = denotes that a language uses either atomic equality or deep equality, no/yes denotes that a language either does not have negation or has negation. The complexity results in the upper part of the table are derived from Koch's results in [Koc06]. All the complexity results in the table are complete except for the case of HOCV^0 and XQ with deep equality, which have a TA$[2^{O(n)}, O(n)]$ lower bound and EXPSPACE upper bound. When the language contains variables of order 1, the complexity of the evaluation problem becomes EXPSPACE-complete. The lower part of Table 5.2 contains our complexity results for evaluating the higher-order extension of Core XQuery and the higher-order complex-valued query language. Similarly to the language for relational databases, restrictions on nesting lead to drastic reductions in the complexity of the evaluation problem.

HOCV	XQH	Negation	Equality	General	Un-nested
HOCV^0	Core XQuery	no	$[=_{atomic}]$	NEXPTIME	undefined
		yes	$[=_{atomic}]$	TA$[2^{O(n)}, O(n)]$	
		no/yes	$[=_{deep}]$	**in** EXPSPACE	
HOCV^{2m-1}	XQH^{2m-1}	no/yes	=	m-EXPSPACE	EXPSPACE
HOCV^{2m}	XQH^{2m}	no/yes	=	(m+1)-EXPTIME	EXPSPACE

Table 5.2: Complexity of evaluation for HOCV and XQH

.

Chapter 6

Containment between Higher-Order Queries

This chapter first defines containment problems for higher-order queries for both the relational case and the complex-valued case. We then consider the complexity of these problems when queries are of order 1 and when queries are of order 2. The containment between order 2 terms is called *higher-order containment*. In the higher-order containment problem, two queries are normalized and take as input both query variables and order 0 variables, which are relational variables or nested relational variables.

For the relational case, we give tight bounds for both containment between order 1 terms and between normalized order 2 terms. The complexity containment between order 1 HO terms will be shown in Theorem 6.6. When terms are in normal form, we show the complexity of the containment and equivalence problems for higher-order queries that manipulate positive Relational Algebra queries. We will show that the complexity of containment between normalized order 2 terms is Π_2^{P}-complete (Theorem 6.8), which is in the same complexity class with the containment between RA^+ expressions. Additionally, we give results when there are integrity constraints. We also give specialized results when omitting query variables in one side, when omitting the union operator, and a number of other cases.

For the complex-valued case, in Theorem 6.29 we give upper bounds and lower bounds for containment between arbitrary order 1 terms. Our complexity results are different from previous work [LS97, DHT04, BMS11] because our languages are different, e.g., our terms contain variables of arbitrary degree. In Theorem 6.31, we also give tight bounds for higher-order containment between normalized order 2 terms, which surprisingly are the same as the bounds for the relational case. The results for containment between normalized order 2 terms are incomparable with previous work because the higher-order containment problem is first time defined.

Table 6.1 summarizes the main results of this chapter. In the table, the containment between higher-order terms is classified by order and degree of the terms. "Ord.", "Deg.", "Norm." and " Unnorm." are abbreviations for "Order", "Degree", "Normalized" and "Unnormalized", respectively. Except for containment between unnormalized HOCVP terms of order 1, all the other complexity results in the table are tight.

Ord.	Deg.	Norm. HO	Unnorm. HO	Norm. HOCVP	Unnorm. HOCVP
1	0	Π_2^{P} [SY80]	co-NEXPTIME [BG10]	Π_2^{P} Theorem 6.30	in TA$(2^{O(n)}, 1)$ Proposition 6.27
	$k-1$	undefined	co-k-NEXPTIME Theorem 6.6	undefined	co-k-NEXPTIME Theorem 6.29
2	1	Π_2^{P} Theorem 6.8	–	Π_2^{P} Theorem 6.31	–
	$k-1$	undefined	–	undefined	–

Table 6.1: Complexity of containment.

6.1 The containment problem

This section introduces the containment problems for the relational case and for the complex-valued case.

The containment problem for the relational case.

We define a generalization of the *containment* relation $\subseteq$ between HO terms.

For terms of order 0, the definition of containment is straightforward: given two closed terms Φ and Φ' of the same relational type, we write $\Phi \subseteq \Phi'$ iff $[\![\Phi]\!]_{\mathcal{I}_0} \subseteq [\![\Phi']\!]_{\mathcal{I}_0}$ with $\mathcal{I}_0$ the underlying interpretation for Φ and Φ'.

We then extend the definition of containment from HO terms of relational type to HO terms of order n with $n > 0$ as follows.

Let Φ and Φ' be two closed HO terms that have normal form after β-reduction $\lambda X_1. \ldots \lambda X_n.\, \varphi$ and $\lambda X_1. \ldots \lambda X_n.\, \varphi'$, respectively. Additionally, suppose Φ and Φ' are of the same type $\mathcal{T}_1 \to \ldots \to \mathcal{T}_n \to \mathcal{T}$ with $\mathcal{T}$ of order 0, and let $\mathcal{I}$ be an interpretation for $X_1, \ldots, X_n$. We write

$$\Phi \subseteq_{\mathcal{I}} \Phi' \quad \text{iff} \quad \forall i \in [1,n]\ \forall x_i \in \mathit{Dom}_{\mathcal{I}}(X_i).\ \Phi(x_1 \ldots x_n) \subseteq \Phi'(x_1 \ldots x_n).$$

As an example, given two order 2 terms Φ and Φ' of the same type, we write $\Phi \subseteq_{\lambda\mathsf{RA}^+} \Phi'$ iff, for all instances $\mathtt{Q}_1, ..., \mathtt{Q}_m, \mathtt{R}_1, ..., \mathtt{R}_n$ of the formal arguments $Q_1, ..., Q_m$, $R_1, ..., R_n$ in Φ and Φ', with each $\mathtt{Q}_i$ ranging over the set of queries of Positive Relational Algebra and each $\mathtt{R}_i$ ranging over the set of finite relations, we have $[\![\Phi]\!](\mathtt{Q}_1, ..., \mathtt{Q}_m, \mathtt{R}_1, ..., \mathtt{R}_n) \subseteq [\![\Phi']\!](\mathtt{Q}_1, ..., \mathtt{Q}_m, \mathtt{R}_1, ..., \mathtt{R}_n)$.

Definition 10 (The containment problem between HO terms)**.** *The* containment problem *for lefthandside terms in $\mathcal{C}$ and righthandside terms in $\mathcal{C}'$, under the interpretation $\mathcal{I}$, consists of deciding, given two terms $\Phi \in \mathcal{C}$ and $\Phi' \in \mathcal{C}'$ of the same type, whether $\Phi \subseteq_{\mathcal{I}} \Phi'$.*

The containment problem for the complex-valued case.

We define containment between HOCV terms of order 0 based on Definition 3.1 in [LS97] of Levy and Suciu. Given two closed HOCV_0 terms Φ and Φ' of the same nested relational type, we define the containment $\Phi \subseteq \Phi'$ inductively as follows.

- If Φ and Φ' are of the same base type, then $\Phi \subseteq \Phi'$ iff $\Phi = \Phi'$.
- If $\Phi = \langle A_1 : \Phi_1, \ldots, A_n : \Phi_n \rangle$ and $\Phi' = \langle A_1 : \Phi'_1, \ldots, A_n : \Phi'_n \rangle$, then $\Phi \subseteq \Phi'$ iff $\quad \forall i. \Phi_i \subseteq \Phi'_i$.
- If $\Phi = \{\Phi_1, \ldots, \Phi_n\}$ and $\Phi' = \{\Phi'_1, \ldots, \Phi'_m\}$, then $\Phi \subseteq \Phi'$ iff $\quad \forall i \exists j. \Phi_i \subseteq \Phi'_j$

We then extend the definition of containment from HOCV terms of order 0 to HOCV terms of order n with $n > 0$ as follows. Let Φ and Φ' be two closed HOCV that have normal form after β-reduction as $\lambda X_1 . \ldots \lambda X_n . \varphi$ and $\lambda X_1 . \ldots \lambda X_n . \varphi'$, respectively. Additionally, suppose Φ and Φ' are of the same type $\mathcal{T}_1 \to \ldots \to \mathcal{T}_n \to \mathcal{T}$ with $\mathcal{T}$ of order 0, and let $\mathcal{I}$ be an interpretation for $X_1, \ldots, X_n$. We write

$$\Phi \subseteq_{\mathcal{I}} \Phi' \quad \text{iff} \quad \forall i \in [1, n]\, \forall x_i \in \mathit{Dom}_{\mathcal{I}}(X_i).\ \Phi(x_1 \ldots x_n) \subseteq \Phi'(x_1 \ldots x_n).$$

We will only study the interpretation $\mathcal{I} = \lambda\mathsf{MA}^+$ for the complex-valued case.

Definition 11 (The containment problem between HOCV terms)**.** *The* containment problem *for lefthandside terms in $\mathcal{C}$ and righthandside terms in $\mathcal{C}'$, consists of deciding, given two terms $\Phi \in \mathcal{C}$ and $\Phi' \in \mathcal{C}'$ of the same type, whether $\Phi \subseteq_{\lambda\mathsf{MA}^+} \Phi'$.*

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* (i.e., given a term Φ, decide whether there is an input x such that $\Phi(x)$ evaluates to `true`) and the *extensional equivalence* (i.e., given Φ and Φ', decide whether $\Phi(x) = \Phi'(x)$ for every input x). As an example, two terms Φ and Φ' are extensionally equivalent, under an underlying interpretation $\mathcal{I}$, iff $\Phi \subseteq_{\mathcal{I}} \Phi'$ and $\Phi' \subseteq_{\mathcal{I}} \Phi$.

We will always consider the computational complexity of our containment problems for HO and for HOCV in terms of the *size* of the terms, as defined earlier in Subsection 4.1.1 and Subsection 4.2.2. When two terms are of order 2, we refer to the containment problem as the *higher-order containment* problem.

We will study these problems 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 range of query variables. In the case of unnormalized terms, the problem has an additional source of complexity, related to the phenomenon of *sharing* subterms during β-reductions; it is exactly the source of complexity that is eliminated in considering normalized terms. For unnormalized terms, we will only consider the containment problem for order 1 terms, which evaluate to ordinary queries. Since these terms take as input a set of order 0 constants, which are clear from the context, we omit the signature in the notation of containment.

In the remainder of this section, we give a number of examples for the containment problem between order 1 terms and between order 2 terms.

Example 11. *As in Example 4, let R, R_1, and R_2 be relational variables with integer attributes (a, b), and τ_p^2 be an ordinary conjunctive query returning paths of length 2.*

Consider the following two terms.

$$\begin{aligned} \tau_p^{16} &= (\lambda \mathbf{Q_1}.\lambda \mathbf{R_1}.(\mathbf{Q_1}(\mathbf{Q_1}(\mathbf{R_1}))))((\lambda \mathbf{Q}.\lambda \mathbf{R}.(\mathbf{Q}(\mathbf{Q}(\mathbf{R}))))\tau_p^2) \\ \tau_p^{8} &= (\lambda \mathbf{Q}.\lambda \mathbf{R}.(\mathbf{Q}(\mathbf{Q}(\mathbf{Q}(\mathbf{R})))))\tau_p^2 \end{aligned}$$

One can check that τ_p^{16} (resp., τ_p^8) takes as input a graph and returns a graph containing all the pairs of nodes having a path of length 16 (resp., 8) between them. This implies that τ_p^{16} is contained in τ_p^8.

The following examples of order 2 containment also show how the higher-order containment may depend on the underlying interpretation for the domains of the query variables.

Example 12. *Let R be a variable of relational type $\mathcal{R} = (a)$, with $Dom(a) = \mathbb{Z}$, and let Q be a variable of query type $\mathcal{R} \to \mathcal{R}$. Consider the order* 2 HO *terms:*

$$\Phi \;= \lambda Q.\, \lambda R.\, Q\left(Q\big(\sigma_{a=1}(R)\big)\right)$$
$$\Phi' \;= \lambda Q.\, \lambda R.\, Q\big(\sigma_{a=1}(R)\big)$$

over the signature CQ*. Take an arbitrary query constant* Q *and an arbitrary relational constant* R *as instances of Q and R. Note that $\sigma_{a=1}(\mathtt{R})$ is either a singleton or the empty set. If a CQ* Q *returns a non-empty relation on input $\sigma_{a=1}(\mathtt{R})$, then it must return a singleton consisting either of the tuple t_1, with $t_1.a = 1$, or the tuple t_2, with $t_2.a = c$, for some constant c that appears in* Q*. Now, if $\mathtt{Q}\big(\sigma_{a=1}(\mathtt{R})\big) = \{t_1\}$, then, by monotonicity, we have $\mathtt{Q}\left(\mathtt{Q}\big(\sigma_{a=1}(\mathtt{R})\big)\right) = \{t_1\}$. Otherwise, if $\mathtt{Q}\big(\sigma_{a=1}(\mathtt{R})\big) = \{t_2\}$, then case analysis on* Q *shows that $\mathtt{Q}\left(\mathtt{Q}\big(\sigma_{a=1}(\mathtt{R})\big)\right)$ must be either the singleton $\{t_2\}$ or the empty set. Therefore, we have that Φ is contained in Φ' under the interpretation of the query variables by Conjunctive Queries, shortly, $\Phi \subseteq_{\lambda\mathsf{CQ}} \Phi'$. On the other hand, we have $\Phi \not\subseteq_{\lambda\mathsf{RA}^+} \Phi'$, since we can take* Q *such that $\mathtt{Q}\big(\{t_1\}\big) = \{t_2\}$ and $\mathtt{Q}\big(\{t_2\}\big) = \{t_3\}$, with $t_1.a = 1$, $t_2.a = 2$, and $t_3.a = 3$.*

Example 13. *Let R be a variable of relational type $\mathcal{R} = (a)$, with $Dom(a) = \mathbb{Z}$, and let Q be a variable of query type $\mathcal{R} \to \{\}$. Consider the order* 2 HO *terms:*

$$\Phi \;= \lambda Q.\, \lambda R.\, \pi_{\emptyset}\big(\sigma_{b=2}\big(Q(\sigma_{a=1}(R))\big)\big) \;\bowtie\; \pi_{\emptyset}\big(\sigma_{b=3}\big(Q(\sigma_{a=1}(R))\big)\big)$$
$$\Phi' \;= \lambda Q.\, \lambda R.\, \pi_{\emptyset}\left(\sigma_{a=1}\big(\sigma_{a=2}(R)\big)\right)$$
$$\textit{(Φ' returns always } \mathtt{false}\textit{)}$$

over the signature CQ*. When we instantiate Q by a CQ* Q*, $\Phi(\mathtt{Q})$ turns out to be unsatisfiable, since for any instance* R *of R, we have $\sigma_{a=1}(\mathtt{R})$ is either a singleton or the empty set and hence $\sigma_{b=2}\big(\mathtt{Q}(\sigma_{a=1}(\mathtt{R}))\big)$ and $\sigma_{b=3}\big(\mathtt{Q}(\sigma_{a=1}(\mathtt{R}))\big)$ cannot return a non-empty set at the same time. However, if we choose $\mathtt{R} = \{t_1\}$ and* Q *to be a union of conjunctive queries in such a way that $\mathtt{Q}(\mathtt{R}) = \{t_2\} \cup \{t_3\}$, where $t_1.a = 1$, $t_2.a = 2$, and $t_3.a = 3$, then $\Phi(\mathtt{Q}, \mathtt{R})$ evaluates to* true*. This shows that $\Phi \subseteq_{\lambda\mathsf{CQ}} \Phi'$ and $\Phi \not\subseteq_{\lambda\mathsf{RA}^+} \Phi'$.*

Example 14. *Let R_1, R_2 be two variables of relational type $\mathcal{R} = (a)$, with $Dom(a) = \mathbb{Z}$, and let Q be a variable of query type $\mathcal{R} \to \{\}$. Consider the order* 2 HO *terms:*

$$\Phi \;= \lambda Q.\, \lambda R_1.\, \lambda R_2.\, Q(R_1)$$
$$\Phi' \;= \lambda Q.\, \lambda R_1.\, \lambda R_2.\, Q(R_1 \cup R_2)$$

over the signature RA^+. *For every monotone query* $\mathtt{Q}$ *(and, in particular, for every query of the Positive Relational Algebra) and for every pair of relations* $\mathtt{R}_1, \mathtt{R}_2$, *we have* $\mathtt{Q}(\mathtt{R}_1) \subseteq \mathtt{Q}(\mathtt{R}_1 \cup \mathtt{R}_2)$. *Thus,* $\Phi \subseteq_{\lambda\mathsf{RA}^+} \Phi'$. *On the other hand, for any signature* $\mathcal{F}$ *that extends* RA^+ *with the difference operator* $\setminus$, *we have* $\Phi \not\subseteq_{\lambda\mathcal{F}} \Phi'$, *since we can choose* $\mathtt{R}_1 = \{t_1\}$, $\mathtt{R}_2 = \{t_2\}$, *with* $t_1.a = 1$ *and* $t_2.a = 2$ *as instances of* R_1, R_2, *and* $\mathtt{Q} = \lambda S.\, \mathtt{true} \setminus \pi_\emptyset\big(\sigma_{a=2}(S)\big)$ *as an instance of* Q.

To finish this definition section, we give a simple example of containment between HOCVP terms.

Example 15. *Let* $\mathtt{C}_1$ *and* $\mathtt{C}_2$ *be two nested relational constant of type* $\mathcal{R} = \{\langle A \rangle\}$, *with* $Dom(A) = \mathbb{Z}$, $\mathtt{C}_1 = \{\langle 1 \rangle, \langle 2 \rangle\}$ *and* $\mathtt{C}_2 = \{\langle 1 \rangle, \langle 2 \rangle, \langle 3 \rangle\}$. *Let* R_1, R_2 *be two variables of nested relational type* $\mathcal{R}$, *and let* Q *be a variable of query type* $\mathcal{R} \to \mathcal{R}$. *Consider the order* 2 HOCVP *terms:*

$$\Phi = \lambda Q.\, \lambda R_1.\, \lambda R_2.\, Q(R_1) \cup Q(R_2) \cup \mathtt{C}_1$$

$$\Phi' = \lambda Q.\, \lambda R_1.\, \lambda R_2.\, Q(R_1 \cup R_2) \cup \mathtt{C}_2$$

It is easy to see that $\Phi \subseteq_{\lambda\mathsf{MA}^+} \Phi'$ *because both* $Q(R_1)$ *and* $Q(R_2)$ *are contained in* $Q(R_1 \cup R_2)$, *and* $\mathtt{C}_1 \subseteq_{\lambda\mathsf{MA}^+} \mathtt{C}_2$.

6.2 Containment between HO terms of order 1

This section examines the containment problem for terms of order 1, that is, terms that evaluate to queries, rather than to query functionals. We omit the interpretation in the notation of containment. Since normalized order 1 HO terms are indeed Relational Algebra expressions, where containment is well-understood, we only study the containment problem for unnormalized terms. We start with terms containing variables of low order, then extend the results to general cases.

6.2.1 Containment between lower degree terms

We now turn to the complexity of the containment problem between unnormalized HO_1 terms of degree 0 and degree 1. Degree 0 terms are defined using λ-abstraction over relational variables only.

Clearly this is undecidable for RA, since even the satisfiability problem is undecidable.

By Proposition 4.1, $\mathsf{HO}_1^0[\mathsf{RA}^+]$ containment is the same as Nonrecursive Datalog containment. From the results in [BG10], we have the following complexity result:

Proposition 6.1. *The problem of deciding the containment* $\Phi \subseteq \Phi'$*, where* $\Phi \in \mathsf{HO}_1^0[\mathsf{RA}^+]$ *and* $\Phi' \in \mathsf{HO}_1^0[\mathsf{RA}^+]$*, is co-*NEXPTIME*-complete.*

We show that containment of $\mathsf{HO}_1^0[\mathsf{CQ}]$ terms in $\mathsf{HO}_1^0[\mathsf{RA}^+]$ terms is in PSPACE:

Proposition 6.2. *The problem of deciding the containment* $\Phi \subseteq \Phi'$*, where* $\Phi \in \mathsf{HO}_1^0[\mathsf{CQ}]$ *and* $\Phi' \in \mathsf{HO}_1^0[\mathsf{RA}^+]$*, is in* PSPACE*.*

Proof. The intuition behind the proof of the proposition is that we can explore the unfolding of Φ in PSPACE. We make this precise by giving canonical names to variables in the unfolding.
Assume that a query $\mathtt{Q}$ is given as a set of rules $Ru_1 \dots Ru_k$ with Ru_i of the form $H_i(\vec{x}) \leftarrow \phi_i(\vec{x})$, where ϕ_i is a CQ mentioning only relations $H_j : j < i$. By a standard transformation [GP03] we can assume that each ϕ_i has only two occurrences of relation symbols in it. Let $[\mathtt{Q}]$ be the unfolding of $\mathtt{Q}$ as a UCQ, obtained by recursively replacing an occurrence of $H_i(\vec{x})$ with $\phi_i(\vec{x})$. A partial unfolding is any intermediate formula resulting from this process. A *name* is a sequence of pairs (i, j) with $i \leq k, j \in \{1, 2\}$ of length at most k. We associate every atom and every variable in a partial unfolding of $[\mathtt{Q}]$ with a name as follows: in the original $\mathtt{Q}$, every atom is associated with the empty name. If in partial unfolding η we replace the j^{th} occurrence O of $H_i(\vec{x})$ in η with $\phi_i(\vec{x})$ to get η', then we associate every atom and also every variable that was introduced in η' with $name(O), (i, j)$. Note that every name is thus associated with at most one relation symbol and many variables. It is easy to show that one can check properties of names in PSPACE.
Our algorithm will now mimic the standard PSPACE algorithm for evaluating a Non-recursive Datalog query $\mathtt{P}$ on an explicitly given database, but instead of guessing elements of the database, it guesses a $\mathtt{Q}$-name. □

Since containment is harder than evaluation, we have that the containment problem of $\mathsf{HO}_1^0[\mathsf{CQ}]$ in $\mathsf{HO}_1^0[\mathsf{RA}^+]$ is PSPACE-complete. More specifically, from the results on the evaluation problem, we can say that the problem is hard even when the left-handside terms are as restricted as possible and the righthandside terms do not use unions:

Corollary 6.3. *The problem of deciding the containment* $\Phi \subseteq \Phi'$*, where* Φ *is a conjunctive query and* $\Phi' \in \mathsf{HO}_1^0[\mathsf{CQ}_\mathsf{C}]$*, is* PSPACE*-hard.*

However, if we restrict the righthandside terms of the containment problem, we do get a better bound for $\mathsf{HO}_1^0[\mathsf{CQ}]$. The argument also uses the idea of compact names, as in Proposition 6.2:

Theorem 6.4. *The problem of deciding the containment $\Phi \subseteq \Phi'$, where $\Phi \in \mathsf{HO}_1^0[\mathsf{CQ}]$ and Φ' is a conjunctive query, is NP-complete.*

A proof for the theorem above is also presented in Proposition 13 in [BG10]. Now, we look at the containment between order 1 degree 1 terms.

Theorem 6.5. *The containment problem between two $\mathsf{HO}_1^1[\mathsf{RA}^+]$ terms (unnormalized order 1 degree 1 terms) is co-2-NEXPTIME-complete.*

Proof. Every degree 1 term can be reduced to a degree 0 term with an exponential blows up. Thus the co-2-NEXPTIME-membership result is directly derived from Proposition 6.2.

To show the hardness of the problem, we reduce the acceptance of a Non-deterministic Turing Machine (NTM) $\mathcal{M}$ which runs in double exponential time to the non-containment problem $\Phi_1 \not\subseteq \Phi_2$. Note that when Φ_1 and Φ_2 are boolean queries, $\Phi_1 \not\subseteq \Phi_2$ iff there exists a database instance such that Φ_1 evaluates to True and Φ_2 evaluates to False.

Let us recall the representation of NTMs used in the proof of Proposition 5.6. An NTM $\mathcal{M}$ is represented as a 5-tuple $(\mathcal{Q}, \Sigma, \Gamma, \delta, q_0, \mathcal{F})$ with:

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

We assume that the input is bounded by n, and $\omega = \omega_0. \ldots .\omega_n$ with $\omega_i \in \Sigma$ the input string. We will also assume that the running time of the NTM is bounded by 2^{2^n}, which is denoted by N.

We define a relation $C(F, P, A, H, B)$, which is similar to the relation $S(\vec{p}, \vec{a}, h, \vec{b})$ in the proof of complexity for evaluation in Proposition 5.6, to code all the configurations of the NTM. A subset of tuples in C codes the tape at a step k. The attribute F of integer type $\mathcal{Z}$ is used to distinguish different configurations at different steps. The other attributes P, A, H, and B respectively represent positions, the tape symbol

at the position, the presence or absence of the head, and the control state. These attributes have domains $\mathcal{Z}$, Γ, $\{0,1\}$, and $\mathcal{Q}$, respectively.

We now turn to defining the two queries. Intuitively, Φ_1 contains conditions that need to be satisfied and Φ_2 contains conditions that must not be satisfied. Below, we will use a notation that mixes our higher-order calculus with Datalog rules – the semantics should be clear. The notation can be efficiently compiled into our language without Datalog, because Datalog rules can be polynomially transformed to HO_1^0 terms.

Let D and Diff be two input relations having two attributes F_1, F_2 with domains $\mathcal{Z}$. A query Q_1 over a relation D is defined using the following Datalog rules:

$$\begin{aligned} Goal(x,y) &\leftarrow D(x,y), \mathsf{Diff}(x,y) \\ Goal(x,z) &\leftarrow D(x,y), D(y,z), \mathsf{Diff}(x,y), \mathsf{Diff}(y,z), \mathsf{Diff}(x,z) \end{aligned}$$

We use a degree 1 term ρ_1 which repeats Q_1 on D an exponential number of times to define $\mathsf{After}_1 = (\rho_1(Q_1))D$. Notice that the output of After_1 contains all the end points of the paths in D that have length at most 2^{2^n} . Then we can define a linear order:

$$\mathsf{After}(x,y) \leftarrow \mathsf{After}_1(c_0,x), \mathsf{After}_1(x,y)$$

with c_0 a constant, and a successor relation:

$$\mathsf{Succ}(x,y) \leftarrow D(x,y), \mathsf{After}(c_0,x), \mathsf{After}(c_0,y)$$

The occurrences of Diff in the rules above make sure that all tuples occurring in Succ and After also occur in Diff.

We also define the last element $\mathsf{Max}(x)$ that repeats the following query a doubly exponential number of times over After then returning the second attribute:

$$Ans(y,z) \leftarrow \mathsf{Succ}(x,y), \mathsf{Succ}(y,z)$$

From the definitions of Succ, After, and Max above, we know that for all $i, j \in (1, N)$ there exists a tuple (f_i, f_j) in the output of After if $i < j$, and there exists (f_i, f_j) in the output of Succ if $j = i + 1$. Let $\mathbf{f_1}$ and $\mathbf{f_N}$ be two constants that respectively represent the first and the last elements in the linear order After. In the rest of the proof, we will always use bold format for constants. We now give conditions to guarantee that $\pi_F(C)$ contains all $\mathbf{f_1}, \ldots, \mathbf{f_N}$, and $\pi_P(\sigma_{F=f_i}(C))$ contains all $\mathbf{f_1}, \ldots, \mathbf{f_N}$ for each $i \in (1, N)$. We first define the following Datalog program $\mathcal{P}_1$:

$$\begin{aligned} Ans(f_i, f_{j+1}, a, h, b) &\leftarrow R(f_i, f_j, a, h, b), \mathsf{Succ}(f_j, f_{j+1}) \\ Ans(f_{i+1}, \mathbf{f_1}, a, h, b) &\leftarrow R(f_i, f_j, a, h, b), \mathsf{Max}(f_j), \mathsf{Succ}(f_i, f_{i+1}) \end{aligned}$$

Let ρ_2 be a degree 1 term that repeats $\mathcal{P}_1$ $N \cdot N$ times. We then define $Q_2 = \sigma_{F=\mathbf{f_N},P=\mathbf{f_N}}\rho_2(\mathcal{P}_1)\{(f_f, f_1, a, h, b)\}$. Note that Q_2 guarantees that the conditions above about the attributes F and P in C are satisfied.

We are now ready to define two terms Φ_1 and Φ_2.

- The higher-order term Φ_1 is defined as below.

$$\Phi_1() \leftarrow Q_2(), Q_3(), C(f, p', a', h, b'), \mathsf{Max}(f), F(b')$$

  with Q_3 the conditions about the input string. Q_3 is defined by going through the tuples in C with attribute F equal to $\mathbf{f_1}$. Assuming a tuple has attribute P equal to i, if $i \in (1, n)$, the attribute A is equal to ω_i, if $i > n$, the attribute A is equal to the blank symbol (denoted by $\square$). The conditions about the head and the initial state are easily defined through the attributes H and B.

- The UCQ Φ_2 is defined to assure that all the configurations in S and all the transitions in C are legal. If one condition is broken, Φ_2 returns True.

  To make sure that there exists exactly one path in the relation After, we add the following conditions to Φ_2:

$$\begin{aligned}\Phi_2() &\leftarrow \mathsf{Diff}(x, x)\\ \Phi_2() &\leftarrow \mathsf{Succ}(x, y), \mathsf{Succ}(x, z), \mathsf{Diff}(y, z)\end{aligned}$$

  This query implies that the paths occurring in After do not create branches.

  From Φ_1, we know the initial configuration is $\mathbf{f_1}$. Assuming the initial state is coded by a constant $\mathbf{q_0}$, we have the following conditions for the initial configuration:

$$\begin{aligned}\Phi_2() &\leftarrow C(\mathbf{f_1}, p, a, h, b), \mathsf{Diff}(b, \mathbf{q_0})\\ \Phi_2() &\leftarrow C(\mathbf{f_1}, p, a, h, b), \mathsf{After}(n, p), \mathsf{Diff}(a, \square)\end{aligned}$$

  A configuration is wrong if there are two different symbols at a position on the tape, or the head is at two different cells, or there are two different states of the same configuration.

$$\begin{aligned}\Phi_2() &\leftarrow C(f, p, a, h, b), C(f, p', a', h', b'), p = p', \mathsf{Diff}(a, a')\\ \Phi_2() &\leftarrow C(f, p, a, h, b), C(f, p', a', h', b'), h = h' = 1, \mathsf{Diff}(p, p')\\ \Phi_2() &\leftarrow C(f, p, a, h, b), C(f, p', a', h', b'), \mathsf{Diff}(b, b')\end{aligned}$$

Additionally, Φ_2 makes sure that all the transitions between configurations are valid by the following Datalog rules.

$$\Phi_2() \leftarrow \mathsf{Succ}(f, f'), \bigwedge_{\forall(b_1,a_1,b_2,a_2,c)\in T} (\text{WrongTrans}(b_1, a_1, b_2, a_2, c, f, f'))$$

where T represents the transition function δ of the NTM, and c is of type $\{\mathbf{l}, \mathbf{r}, \mathbf{c}\}$, which contains three integer constants and represents the movements of the head of the NTM.

The query WrongTrans$(b_1, a_1, b_2, a_2, c, f, f')$ is the union of the following queries, which give the conditions for the transition represented by the tuple (b_1, a_1, b_2, a_2, c) in T.

- First, we give conditions for the position of the head.

$$\begin{aligned} Ans() &\leftarrow T(b_1, a_1, b_2, a_2, \mathbf{r}), C(f, p, a, 1, b), C(f', p', a', 0, b'), \mathsf{Succ}(p, p') \\ Ans() &\leftarrow T(b_1, a_1, b_2, a_2, \mathbf{l}), C(f, p, a, 1, b), C(f', p', a', 0, b'), \mathsf{Succ}(p', p) \\ Ans() &\leftarrow T(b_1, a_1, b_2, a_2, \mathbf{s}), C(f, p, a, 1, b), C(f', p', a', 0, b'), p = p' \end{aligned}$$

- Second, conditions about the symbol at the position of the head are given by the following query.

$$Ans() \leftarrow T(b_1, a_1, b_2, a_2, c), C(f, p, a_1, 1, b_1), C(f', p, a', h', b'), \mathsf{Diff}(a', a_2)$$

- Third, the following query gives conditions about the new state.

$$Ans() \leftarrow T(b_1, a_1, b_2, a_2, c), C(f', p', a', h', b'), \mathsf{Diff}(b', b_2)$$

- Lastly, we give a query about the cells which do not change their symbols due to the transition.

$$Ans() \leftarrow C(f, p, a, h, b), C(f', p', a', h', b'), h = 0, p = p', \mathsf{Diff}(a, a')$$

It is easy to show that $\Phi_1 \not\subseteq_{\lambda\mathsf{RA}^+} \Phi_2$ iff $\mathcal{M}$ accepts ω within 2^{2^n} steps. Thus, the non-containment problem is 2-NEXPTIME-hard. □

Overall, the containment problem between two order 1 degree 1 terms is co-2-NEXPTIME-complete.

*Note: This method can be adapted to show that the containment between NRDL programs is co-*NEXPTIME*-hard [BG10]. In that case we generate paths of length at most 2^n in* After*.*

6.2.2 Containment between higher degree terms

Theorem 6.6. *The containment problem between two* $\mathsf{HO}_1^{k-1}[\mathsf{RA}^+]$ *terms is co-k-NEXPTIME-complete.*

Proof. The membership is easily shown by reducing degree $(k-1)$ terms to degree 0 terms.

The hardness result is shown by adapting the hardness proof of Theorem 6.5. We use degree $(k-1)$ terms to generate After containing paths of length at most k-hyperexponential in the input size. This can be done using the coding in Proposition 5.12. The other part of the reduction is exactly as in the proof of Theorem 6.5.

Thus the acceptance of a NTM with 2_k^n running time is reducible to the non-containment problem between two degree $(k-1)$ terms. □

Proposition 6.7. *The containment problem for* HO *terms has non-elementary complexity.*

6.3 Containment between normalized HO terms of order 2

The goal of this section will be to prove tight bounds on the complexity of the containment problem for order 2 terms in normal form, namely, for higher-order queries, where the formal arguments (i.e., the query variables and the relational variables) are interpreted by terms of the positive Relational Algebra (RA^+).

6.3.1 Complexity of higher-order containment

The goal of this subsection is to prove:

Theorem 6.8. *The problem of deciding the containment* $\Phi \subseteq_{\lambda\mathsf{RA}^+} \Phi'$*, where* $\Phi, \Phi' \in \mathsf{HO}_2^{\downarrow}[\mathsf{RA}^+]$*, is* Π_2^{P}*-complete.*

We will need to build up a bit of infrastructure first. We start by introducing some variants of the classical problem of deciding containment of CQs in UCQs. The main variation is that containment is relative to a set of constraints of the form $\mathtt{R}_i \subseteq \mathtt{R}_j$ (positive constraints) or $\mathtt{R}_i \nsubseteq \mathtt{R}_j$ (negative constraints), where $\mathtt{R}_i$ and $\mathtt{R}_j$ are relational symbols. Moreover, we introduce a disjunctive variant of the constrained containment problem.

Definition 12. *Below are two variants of the relational containment problems.*

- Constrained Containment Problem: *given two queries* $\mathtt{Q}, \mathtt{Q}'$ *of the same type* $\bar{\mathcal{R}} \to \mathcal{S}$ *and given a set* Σ *of constraints over appropriate relations for* $\bar{\mathcal{R}}$*, the problem consists of deciding whether* $[\![\mathtt{Q}]\!](\bar{\mathtt{R}}) \subseteq [\![\mathtt{Q}']\!](\bar{\mathtt{R}})$ *holds for all instances* $\bar{\mathtt{R}}$ *satisfying the constraints in* Σ*;*

- Constrained Disjunctive Containment Problem: *given some queries* $\mathtt{Q}_1, ..., \mathtt{Q}_n$ *and* $\mathtt{Q}'_1, ..., \mathtt{Q}'_n$*, having types* $\bar{\mathcal{R}} \to \mathcal{S}_1$*, ...,* $\bar{\mathcal{R}} \to \mathcal{S}_n$*, and given a set* Σ *of constraints over appropriate relations for* $\bar{\mathcal{R}}$*, the problem consists of deciding whether, for every instance* $\bar{\mathtt{R}}$ *satisfying* Σ*, there is an index* $1 \leq i \leq n$ *such that* $[\![\mathtt{Q}_i]\!](\bar{\mathtt{R}}) \subseteq [\![\mathtt{Q}'_i]\!](\bar{\mathtt{R}})$ *holds.*

If the set Σ of constraints in the above definition is not specified (or it always evaluates to `true`), then the two problems are simply called *containment problem* and *disjunctive containment problem*. Note that the (constrained) disjunctive containment problem is more general than the (constrained) containment problem.

The first ingredient of the proof of Theorem 6.8 is the following proposition.

Proposition 6.9. *The disjunctive containment problem for lefthandside CQs and righthandside* RA^+*-queries, under positive and negative containment constraints, is NP-complete.*

Before showing the proposition above, we first show:

Lemma 3. *The disjunctive containment problem for lefthandside CQs and righthandside* RA^+*-queries, under positive and negative containment constraints, can be reduced in polynomial time to the containment problem for lefthandside CQs and righthandside* RA^+*-queries, under positive containment constraints.*

Proof. Let us fix a tuple $\bar{\mathcal{R}} = \mathcal{R}_1 \times ... \times \mathcal{R}_n$ of relational types and an additional relational type $\mathcal{S}$. We consider some CQs $\mathtt{Q}_1, ..., \mathtt{Q}_m$ of type $\bar{\mathcal{R}} \to \mathcal{S}$, some RA^+-queries $\mathtt{Q}'_1, ..., \mathtt{Q}'_m$ of the same type $\bar{\mathcal{R}} \to \mathcal{S}$, and a set Σ of positive and negative containment constraints over objects of type $\bar{\mathcal{R}}$. We will reduce the given instance of the constrained disjunctive containment problem to an equivalent instance of the unconstrained disjunctive-free containment problem over boolean queries. To do that, we need to "hide" the output of the queries $\mathtt{Q}_1, \mathtt{Q}'_1, ..., \mathtt{Q}_m, \mathtt{Q}'_m$ under an expanded input structure. We thus introduce types $\bar{\mathcal{S}}$ and $\bar{\mathcal{T}}$ as follows. $\bar{\mathcal{S}}$ is simply the m-fold cartesian product $\mathcal{S}^m$ of $\mathcal{S}$. We first need to partition the set Σ of containment constraints into two subsets, Σ_+ and Σ_-, that contain the positive and the negative containment constraints, respectively. We then enumerate the negative containment

constraints as follows: $\Sigma_- = \{R_{i_1} \not\subseteq R_{j_1}, ..., R_{i_l} \not\subseteq R_{j_l}\}$, where $l = |\Sigma_-|$ and i_k, j_k, for $1 \leq k \leq l$, range over $\{1, ..., n\}$. Clearly, the presence of a constraint of the form $R_{i_k} \not\subseteq R_{j_k}$ witnesses the fact that the types $\mathcal{R}_{i_k}$ and $\mathcal{R}_{j_k}$ coincide; we shortly denote such a type by $\mathcal{T}_k$. Finally, we define $\bar{\mathcal{T}} = \mathcal{T}_1 \times ... \times \mathcal{T}_l$.

We are now ready to explain the reduction. We first transform the queries $\mathtt{Q}_1, \mathtt{Q}'_1, ..., \mathtt{Q}_m, \mathtt{Q}'_m$ into equivalent boolean queries $\mathtt{B}_1, \mathtt{B}'_1, ..., \mathtt{B}_m, \mathtt{B}'_m$ of type $(\bar{\mathcal{R}} \times \bar{\mathcal{S}} \times \bar{\mathcal{T}}) \to \{\}$ (recall that $\{\}$ is the empty type, which can be used to model boolean queries). Formally, for every index $1 \leq i \leq m$, we define:

$$\begin{aligned} \mathtt{B}_i &= \lambda \bar{R}.\, \lambda \bar{S}.\, \lambda \bar{T}.\ \pi_\emptyset\big(\mathtt{Q}_i(\bar{R}) \cap S_i\big) \\ \mathtt{B}'_i &= \lambda \bar{R}.\, \lambda \bar{S}.\, \lambda \bar{T}.\ \pi_\emptyset\big(\mathtt{Q}'_i(\bar{R}) \cap S_i\big) \end{aligned}$$

where $\bar{R} = (R_1, ...R_n)$ ranges over objects of type $\bar{\mathcal{R}}$, $\bar{S} = (S_1, ...S_m)$ ranges over objects of type $\bar{\mathcal{S}}$, and $\bar{T} = (T_1, ...T_l)$ ranges over objects of type $\bar{\mathcal{T}}$ (note that the queries $\mathtt{B}_i$ and $\mathtt{B}'_i$ do not depend on the argument $\bar{T}$, which will be used later). Given the above definition, we have that $\mathtt{B}_i(\bar{R}, \bar{S}, \bar{T}) = \mathtt{true}$ (resp., $\mathtt{B}'_i(\bar{R}, \bar{S}, \bar{T}) = \mathtt{true}$) iff S_i has non-empty intersection with $\mathtt{Q}_i(\bar{R})$ (resp., $\mathtt{Q}'_i(\bar{R})$) and hence $\mathtt{Q}_i(\bar{R}) \subseteq \mathtt{Q}'_i(\bar{R})$ holds iff, for every $\bar{S}$ of type $\bar{\mathcal{S}}$ and every $\bar{T}$ of type $\bar{\mathcal{T}}$, we have $\mathtt{B}_i(\bar{R}, \bar{S}, \bar{T}) \to \mathtt{B}'_i(\bar{R}, \bar{S}, \bar{T})$. From this property, we obtain the following reduction:

$$\forall\ \bar{R} \vDash \Sigma \bigvee_{1 \leq i \leq m} \big(\mathtt{Q}_i(\bar{R}) \subseteq \mathtt{Q}'_i(\bar{R})\big)$$

iff $$\forall\ \bar{R} \vDash \Sigma \bigvee_{1 \leq i \leq m} \forall\ S_i : \mathcal{S}\ \big(S_i \cap \mathtt{Q}_i(\bar{R}) \neq \emptyset \to S_i \cap \mathtt{Q}'_i(\bar{R}) \neq \emptyset\big)$$

iff $$\forall\ \bar{R} \vDash \Sigma,\ \bar{S} : \bar{\mathcal{S}},\ \bar{T} : \bar{\mathcal{T}} \bigvee_{1 \leq i \leq m} \big(\mathtt{B}_i(\bar{R}, \bar{S}, \bar{T}) \to \mathtt{B}'_i(\bar{R}, \bar{S}, \bar{T})\big)$$

iff $$\forall\ \bar{R} \vDash \Sigma,\ \bar{S} : \bar{\mathcal{S}},\ \bar{T} : \bar{\mathcal{T}} \bigvee_{1 \leq i \leq m} \neg \mathtt{B}_i(\bar{R}, \bar{S}, \bar{T}) \ \vee \bigvee_{1 \leq i \leq m} \mathtt{B}'_i(\bar{R}, \bar{S}, \bar{T}).$$

Finally, we show how to get rid of the negative containment constraints by translating them into appropriate query containment relations. We recall the enumeration $\{R_{i_1} \not\subseteq R_{j_1}, ..., R_{i_l} \not\subseteq R_{j_l}\}$ of Σ_- and, for every index $1 \leq k \leq l$, we introduce two boolean queries $\mathtt{C}_k, \mathtt{C}'_k$ of type $(\bar{\mathcal{R}} \times \bar{\mathcal{S}} \times \bar{\mathcal{T}}) \to \{\}$:

$$\begin{aligned} \mathtt{C}_k(\bar{R}, \bar{S}, \bar{T}) &= \pi_\emptyset\big(R_{i_k} \cap T_k\big) \\ \mathtt{C}'_k(\bar{R}, \bar{S}, \bar{T}) &= \pi_\emptyset\big(R_{j_k} \cap T_k\big) \end{aligned}$$

(note that the above queries do not depend on the argument $\bar{S}$). Now, consider an object $\bar{R} = (R_1, ..., R_n)$ of type $\bar{\mathcal{R}}$. We have that $\bar{R}$ *violates* the negative constraint

$R_{i_k} \not\subseteq R_{j_k}$ iff, for every $\bar{S}$ of type $\bar{\mathcal{S}}$ and every $\bar{T}$ of type $\bar{\mathcal{T}}$, $\mathtt{C}_k(\bar{R}, \bar{S}, \bar{T})$ implies $\mathtt{C}'_k(\bar{R}, \bar{S}, \bar{T})$. This argument shows that

$$\forall\ \bar{R} \vDash \Sigma,\ \bar{S} : \bar{\mathcal{S}},\ \bar{T} : \bar{\mathcal{T}}$$

$$\bigvee_{1 \le i \le m} \neg \mathtt{B}_i(\bar{R}, \bar{S}, \bar{T}) \ \vee \bigvee_{1 \le i \le m} \mathtt{B}'_i(\bar{R}, \bar{S}, \bar{T})$$

iff $\forall\ \bar{R} \vDash \Sigma_+,\ \bar{S} : \bar{\mathcal{S}},\ \bar{T} : \bar{\mathcal{T}}$

$$\bigvee_{1 \le i \le m} \neg \mathtt{B}_i(\bar{R}, \bar{S}, \bar{T}) \ \vee \bigvee_{1 \le i \le m} \mathtt{B}'_i(\bar{R}, \bar{S}, \bar{T}) \ \vee \bigvee_{1 \le k \le l} \big(R_{i_k} \subseteq R_{j_k}\big)$$

iff $\forall\ \bar{R} \vDash \Sigma_+,\ \bar{S} : \bar{\mathcal{S}},\ \bar{T} : \bar{\mathcal{T}}$

$$\bigvee_{1 \le i \le m} \neg \mathtt{B}_i(\bar{R}, \bar{S}, \bar{T}) \ \vee \bigvee_{1 \le i \le m} \mathtt{B}'_i(\bar{R}, \bar{S}, \bar{T}) \ \vee \bigvee_{1 \le k \le l} \big(\mathtt{C}_k(\bar{R}, \bar{S}, \bar{T}) \to \mathtt{C}'_k(\bar{R}, \bar{S}, \bar{T})\big)$$

iff $\forall\ \bar{R} \vDash \Sigma_+,\ \bar{S} : \bar{\mathcal{S}},\ \bar{T} : \bar{\mathcal{T}}$

$$\bigvee_{1 \le i \le m} \neg \mathtt{B}_i(\bar{R}, \bar{S}, \bar{T}) \ \vee \bigvee_{1 \le i \le m} \mathtt{B}'_i(\bar{R}, \bar{S}, \bar{T}) \ \vee \bigvee_{1 \le k \le l} \neg \mathtt{C}_k(\bar{R}, \bar{S}, \bar{T}) \ \vee \bigvee_{1 \le k \le l} \mathtt{C}'_k(\bar{R}, \bar{S}, \bar{T})$$

iff $\forall\ \bar{R} \vDash \Sigma_+,\ \bar{S} : \bar{\mathcal{S}},\ \bar{T} : \bar{\mathcal{T}}$

$$\Big(\bigwedge_{1 \le i \le m} \mathtt{B}_i(\bar{R}, \bar{S}, \bar{T}) \ \wedge \bigwedge_{1 \le k \le l} \mathtt{C}_k(\bar{R}, \bar{S}, \bar{T})\Big) \ \to \ \Big(\bigvee_{1 \le i \le m} \mathtt{B}'_i(\bar{R}, \bar{S}, \bar{T}) \ \vee \bigvee_{1 \le k \le l} \mathtt{C}'_k(\bar{R}, \bar{S}, \bar{T})\Big).$$

The latter property can be viewed as an instance of the containment problem for lefthandside CQs and righthandside RA^+-queries, under positive containment constraints. □

We now turn towards proving NP membership for the constrained containment problem:

Lemma 4. *The containment problem for lefthandside CQs and righthandside* RA^+*-queries, under positive containment constraints, is in NP.*

Proof. We give a proof in the case of boolean queries (the proof for the general case can be easily devised from this one). As a preliminary step, we recall the notion of *canonical model* of a boolean CQ $\mathtt{B}$ of type $(\mathcal{R}_1 \times ... \times \mathcal{R}_n) \to \{\}$. This is defined as the tuple $\bar{\mathtt{R}}^{\mathtt{B}} = (\mathtt{R}_1^{\mathtt{B}}, ..., \mathtt{R}_n^{\mathtt{B}})$, where each $\mathtt{R}_i^{\mathtt{B}}$ is a relation of type $\mathcal{R}_i$ that consists of all and only the records $t = (x_1, ..., x_{|\mathcal{R}_i|})$, with x_j being either a variable or a constant, such that $R_i(t)$ appears as a conjunct in $\mathtt{B}$. We recall the well-known characterization of Chandra and Merlin [CM77]:

$$\begin{array}{l} \textit{For every tuple } \bar{\mathtt{R}} \textit{ of relations, } \mathtt{B}(\bar{\mathtt{R}}) = \mathtt{true} \\ \textit{iff there is a homomorphism from } \bar{\mathtt{R}}^{\mathtt{B}} \textit{ to } \bar{\mathtt{R}}. \end{array} \tag{6.1}$$

We now fix a generic instance of the constrained containment problem, which consists of a boolean CQ $\mathtt{B}$ of type $(\mathcal{R}_1 \times ... \times \mathcal{R}_n) \to \{\}$, a boolean RA^+-query $\mathtt{B}'$ of the same type, and a set Σ of positive containment constraints over objects $\bar{R}$ of type $\bar{\mathcal{R}} = \mathcal{R}_1 \times ... \times \mathcal{R}_n$. We transform $\mathtt{B}$ into a new query $\tilde{\mathtt{B}}$ by applying a variant of the chase procedure for inclusion dependencies [AHV95]. Formally, $\tilde{\mathtt{B}}$ is obtained by expanding $\mathtt{B}$ with new conjuncts of the form $R_j(t)$, where t is a tuple that maps attributes of $\mathcal{R}_i$ to variables and constants, whenever $R_i \subseteq R_j$ is a constraint in Σ and $R_i(t)$ appears as a conjunct in the current expansion of $\mathtt{B}$. Note that both the expansion $\tilde{\mathtt{B}}$ and its canonical model $\bar{\mathtt{R}}^{\tilde{\mathtt{B}}}$ have size polynomial in the size of $\mathtt{B}$. We are now able to prove a reduction from the constrained containment problem to a query evaluation problem:

$$\mathtt{B} \subseteq_\Sigma \mathtt{B}' \quad \text{iff} \quad \mathtt{B}'(\bar{\mathtt{R}}^{\tilde{\mathtt{B}}}) = \mathtt{true}.$$

$\Rightarrow$) Assume that $\mathtt{B} \subseteq_\Sigma \mathtt{B}'$. By definition, the canonical model $\bar{\mathtt{R}}^{\tilde{\mathtt{B}}}$ satisfies $\tilde{\mathtt{B}}$, and hence $\mathtt{B}$ as well. Given the definition of $\tilde{\mathtt{B}}$, we have that $\bar{\mathtt{R}}^{\tilde{\mathtt{B}}}$ satisfies also every containment constraint in Σ. Thus, knowing that $\mathtt{B} \subseteq_\Sigma \mathtt{B}'$, we immediately derive $\mathtt{B}'(\bar{\mathtt{R}}^{\tilde{\mathtt{B}}}) = \mathtt{true}$.

$\Leftarrow$) Assume that $\mathtt{B}'(\bar{\mathtt{R}}^{\tilde{\mathtt{B}}}) = \mathtt{true}$. From (6.1), we know that there is a homomorphism h' from the canonical model $\bar{\mathtt{R}}^{\mathtt{B}'}$ of $\mathtt{B}'$ to the canonical model $\bar{\mathtt{R}}^{\tilde{\mathtt{B}}}$ of $\tilde{\mathtt{B}}$. Consider a tuple $\bar{\mathtt{R}}$ of relations that satisfy both the query $\mathtt{B}$ and the containment constraints in Σ. Clearly, by definition of $\tilde{\mathtt{B}}$, we have $\tilde{\mathtt{B}}(\bar{\mathtt{R}}) = \mathtt{true}$. Again from (6.1), we know that there is another homomorphism h from $\bar{\mathtt{R}}^{\tilde{\mathtt{B}}}$ to $\bar{\mathtt{R}}$. The functional composition $h \circ h'$ is a homomorphism from $\bar{\mathtt{R}}^{\mathtt{B}'}$ to $\bar{\mathtt{R}}$. Therefore, by applying again (6.1), we conclude that $\mathtt{B}'(\bar{\mathtt{R}}) = \mathtt{true}$.

The above property immediately yields a non-deterministic polynomial-time algorithm that decides whether $\mathtt{B} \subseteq_\Sigma \mathtt{B}'$. Indeed, it is sufficient to exploit Lemma 5 to non-deterministically guess (i) a conjunct $\tilde{\mathtt{B}}'_i$ of the flattening $\tilde{\mathtt{B}}' = \tilde{\mathtt{B}}'_1 \vee ... \vee \tilde{\mathtt{B}}'_N$ of $\mathtt{B}'$ and (ii) a homomorphism from $\bar{\mathtt{R}}^{\tilde{\mathtt{B}}'_i}$ to $\bar{\mathtt{R}}^{\tilde{\mathtt{B}}}$ (this would witness $\mathtt{B}'(\bar{\mathtt{R}}^{\tilde{\mathtt{B}}}) = \mathtt{true}$ and hence $\mathtt{B} \subseteq_\Sigma \mathtt{B}'$). □

We can now finish the proof of Proposition 6.9:

Proof. NP-hardness follows trivially since the considered constrained disjunctive containment problem includes all instances of the containment problem for CQs, which is known to be NP-hard [AHV95]. As for membership in NP, this follows from Lemma 3 and Lemma 4. □

We will also need some basic facts about the transformation of a given RA^+-query 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 of the Relational Algebra. Formally, the transformation rules are as follows:

$$\begin{aligned}
\rho_{\{a/b\}}(\mathtt{Q}_1 \cup \mathtt{Q}_2) &\rightsquigarrow \rho_{\{a/b\}}(\mathtt{Q}_1) \cup \rho_{\{a/b\}}(\mathtt{Q}_2) \\
\sigma_c(\mathtt{Q}_1 \cup \mathtt{Q}_2) &\rightsquigarrow \sigma_c(\mathtt{Q}_1) \cup \sigma_c(\mathtt{Q}_2) \\
\pi_A(\mathtt{Q}_1 \cup \mathtt{Q}_2) &\rightsquigarrow \pi_A(\mathtt{Q}_1) \cup \pi_A(\mathtt{Q}_2) \\
(\mathtt{Q}_1 \cup \mathtt{Q}_2) \bowtie \mathtt{Q}_3 &\rightsquigarrow (\mathtt{Q}_1 \bowtie \mathtt{Q}_3) \cup (\mathtt{Q}_2 \bowtie \mathtt{Q}_3) \\
\mathtt{Q}_1 \bowtie (\mathtt{Q}_2 \cup \mathtt{Q}_3) &\rightsquigarrow (\mathtt{Q}_1 \bowtie \mathtt{Q}_2) \cup (\mathtt{Q}_1 \bowtie \mathtt{Q}_3).
\end{aligned}$$

By repeatedly applying these rules, one can transform any RA^+-query $\mathtt{Q}$ into an equivalent union of conjunctive queries of the form $\tilde{\mathtt{Q}} = \tilde{\mathtt{Q}}_1 \cup ... \cup \tilde{\mathtt{Q}}_N$, the *flattening* of $\mathtt{Q}$, where N is bounded by an exponential in the size $|\mathtt{Q}|$ of $\mathtt{Q}$ and $\tilde{\mathtt{Q}}_1, ..., \tilde{\mathtt{Q}}_N$ are conjunctive queries of size at most $|\mathtt{Q}|$. The following simple lemma shows that the problem of checking whether a given conjunctive query appears in the flattening of an RA^+-query is in NP.

Lemma 5. *The problem of deciding, given an RA^+-query $\mathtt{Q}$ and a CQ $\mathtt{Q}'$, whether $\mathtt{Q}'$ appears as a conjunct in the flattening $\tilde{\mathtt{Q}} = \tilde{\mathtt{Q}}_1 \cup ... \cup \tilde{\mathtt{Q}}_N$ of $\mathtt{Q}$ is in NP.*

Proof. Let us fix an RA^+-query $\mathtt{Q}$ in normal form and let $\tilde{\mathtt{Q}} = \tilde{\mathtt{Q}}_1 \cup ... \cup \tilde{\mathtt{Q}}_N$ be its flattening. We identify terms with their syntactic trees. By construction, every conjunct $\tilde{\mathtt{Q}}_i$ of $\tilde{\mathtt{Q}}$ can be obtained by removing from the syntactic tree of $\mathtt{Q}$ all $\cup$-labeled nodes and by properly choosing one of the two subtrees issued from each of these nodes. This gives a non-deterministic polynomial-time algorithm that checks whether $\mathtt{Q}' = \tilde{\mathtt{Q}}_i$ for some $1 \leq i \leq N$. □

Now, it is convenient to generalize the containment relation to tuples of relations: given two tuples of relations $\bar{\mathtt{R}} = (\mathtt{R}_1, ..., \mathtt{R}_m)$ and $\bar{\mathtt{R}}' = (\mathtt{R}'_1, ..., \mathtt{R}'_m)$ of the same types, we write $\bar{\mathtt{R}} \subseteq \bar{\mathtt{R}}'$ iff $\mathtt{R}_i \subseteq \mathtt{R}'_i$ holds for all indices $1 \leq i \leq m$. Hereafter, we say that a query $\mathtt{Q}$ is *monotone* iff, for every tuple $\bar{\mathtt{R}} = (\mathtt{R}_1, ..., \mathtt{R}_m)$ and $\bar{\mathtt{R}}' = (\mathtt{R}'_1, ..., \mathtt{R}'_m)$ of relations of appropriate types, $\bar{\mathtt{R}} \subseteq \bar{\mathtt{R}}'$ implies $\mathtt{Q}(\bar{\mathtt{R}}) \subseteq \mathtt{Q}(\bar{\mathtt{R}}')$.

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 6.10. *Fix $m > 0$ and, for all $1 \leq i \leq m$, let $\bar{\mathcal{S}} \to \mathcal{T}_i$ be an order 1 query type. Moreover, fix $k > 0$ and, for all $1 \leq j \leq k$, let (i) i_j be an index from $\{1, ..., m\}$, (ii) $\bar{\mathtt{S}}_j$ be a tuple of relations of types in $\bar{\mathcal{S}}$, and (iii) $\mathtt{T}_j$ be a relation of type $\mathcal{T}_{i_j}$. The following properties are equivalent:*

1. *there exist some RA^+-queries (or, equivalently, some UCQs) $\mathtt{Q}_1, ..., \mathtt{Q}_m$ such that $\mathtt{Q}_{i_j}(\bar{\mathtt{S}}_j) = \mathtt{T}_j$ for all $j \in \{1, ..., k\}$;*
2. *for every pair of indices $j, j' \in \{1, ..., k\}$, if $i_j = i_{j'}$ and $\bar{\mathtt{S}}_j \subseteq \bar{\mathtt{S}}_{j'}$, then $\mathtt{T}_j \subseteq \mathtt{T}_{j'}$.*

Proof. The implication from 1. to 2. is trivial from the monotonicity of RA^+-queries and UCQs. The implication from 2. to 1. is proved as follows. First, we introduce, for every index $j \in \{1, ..., k\}$, a UCQ $\mathtt{Q}^{(j)}$ that, given a tuple $\bar{\mathtt{R}}$ of input relations, returns either $\mathtt{T}_j$ or the empty relation, depending on whether or not the tuple $\bar{\mathtt{S}}_j$ is contained in the tuple $\bar{\mathtt{R}}$. Note that, by construction, we have $\mathtt{Q}^{(j)}(\bar{\mathtt{S}}_j) = \mathtt{T}_j$. We then define the UCQs $\mathtt{Q}_1, ..., \mathtt{Q}_m$ as follows. For every $i^* \in \{1, ..., m\}$, $\mathtt{Q}_{i^*}$ is the union of the conjunctive queries $\mathtt{Q}^{(j)}$ over all indices j such that $i_j = i^*$. It is easy to check that property 2. implies $\mathtt{Q}_{i_j}(\bar{\mathtt{S}}_j) = \mathtt{T}_j$ for all $j \in \{1, ..., k\}$. □

Note: This result depends heavily on the presence of data constants. Characterizations of query definability with constant-free languages do exist — in the database community these date back to the work of Bancilhon [Ban78] and Paredaens [Par78] (see also the recent work of Fletcher et al. [FGPG09], whose results bear some similarity to the proposition above). However such characterizations are more complex, and thus query definability in these other languages can not be reduced to a set of inclusion constraints.

We are now ready to prove that the higher-order containment problem is in Π_2^{P}.

Proposition 6.11. *The problem of deciding the containment $\Phi \subseteq_{\lambda\mathsf{RA}^+} \Phi'$, where $\Phi, \Phi' \in \mathsf{HO}_2^{\downarrow}[\mathsf{RA}^+]$, is in Π_2^{P}.*

Proof. We fix two order 2 terms in normal form

$$\Phi \;=\; \lambda Q_1 \,...\, \lambda Q_m.\; \lambda R_1 \,...\, \lambda R_n.\; \tau$$

$$\Phi' \;=\; \lambda Q_1 \,...\, \lambda Q_m.\; \lambda R_1 \,...\, \lambda R_n.\; \tau'$$

where each Q_i is an order 1 query variable, each R_j is a relational variable, and τ, τ' are well-typed terms of order 0 over the variables $Q_1, ..., Q_m$, $R_1, ..., R_n$ and the constants from the signature RA^+. Below, we provide a logical characterization of

the non-containment relationship $\Phi \not\sqsubseteq_{\lambda\mathsf{RA}^+} \Phi'$, that is, the existence of some queries $\mathtt{Q}_1, ..., \mathtt{Q}_m$ of the positive Relational Algebra and some relations $\mathtt{R}_1, ..., \mathtt{R}_n$ that witness $[\![\tau]\!](\bar{\mathtt{Q}}, \bar{\mathtt{R}}) \not\sqsubseteq [\![\tau']\!](\bar{\mathtt{Q}}, \bar{\mathtt{R}})$.

We start by introducing new relations for the intermediate outputs produced by the subterms of τ and τ' (we explain the construction for τ only, the one for τ' is similar). We enumerate all occurrences of proper subterms of τ that are *arguments to a query variable* Q_i, for some $1 \leq i \leq m$. Let $\sigma_1, ..., \sigma_k$ be such an enumeration. Without loss of generality, we can assume that $j < j'$ holds whenever σ_j occurs inside $\sigma_{j'}$ (note that we distinguish between possible multiple occurrences of the same subterm). We then associate with each occurrence σ_j the following objects: (i) the index $i_j \in \{1, ..., m\}$ of the query variable to which σ_j is applied, (ii) two relations $\mathtt{S}_j, \mathtt{T}_j$ (of appropriate types), (iii) a term $\mathtt{P}_j$ obtained from σ_j by replacing any top-level subterm of the form $Q_{i'}(\sigma_{j'})$ by $\mathtt{T}_{j'}$. We further introduce an additional query constant $\mathtt{P}_0$, obtained from τ by replacing any top-level subterm of the form $Q_{i'}(\sigma_{j'})$ by $\mathtt{T}_{j'}$. Note that, since τ is in normal form, all its subterms are applied to query variables and query constants only. This means that each term $\mathtt{P}_j$, with $0 \leq j \leq k$, is an RA^+-query over the relations $\mathtt{R}_1, ..., \mathtt{R}_m, \mathtt{T}_1, ..., \mathtt{T}_k$. Analogous definitions are given for the objects i'_j, $\mathtt{S}'_j$, $\mathtt{T}'_j$, $\mathtt{P}'_j$ with respect to the occurrences of subterms in τ'.

We can now reduce the non-containment relationship $\Phi \not\sqsubseteq \Phi'$ to the following property (for the sake of brevity, we use the shorthands $\bar{\mathtt{R}} = (\mathtt{R}_1, ..., \mathtt{R}_n)$, $\bar{\mathtt{S}} = (\mathtt{S}_0, ..., \mathtt{S}_k)$, etc.):

$$\begin{aligned}
&\exists\, Q_1, ..., Q_m \\
&\exists\, \bar{R}, \bar{S}, \bar{T}, \bar{S}', \bar{T}'. \quad \mathtt{P}_0(\bar{R}, \bar{T}, \bar{T}') \not\sqsubseteq \mathtt{P}'_0(\bar{R}, \bar{T}, \bar{T}') \;\wedge \\
&\bigwedge_{1 \leq j \leq k} \mathtt{P}_j(\bar{R}, \bar{T}, \bar{T}') = S_j \;\wedge \bigwedge_{1 \leq j \leq h} \mathtt{P}'_j(\bar{R}, \bar{T}, \bar{T}') = S'_j \;\wedge \\
&\bigwedge_{1 \leq j \leq k} Q_{i_j}(S_j) = T_j \;\wedge \bigwedge_{1 \leq j \leq h} Q_{i'_j}(S'_j) = T'_j.
\end{aligned} \tag{6.2}$$

By exploiting Proposition 6.10, we can get rid of the existential quantification over $Q_1, ..., Q_m$ thus obtaining:

$$\begin{aligned}
&\exists\, \bar{R},\, \bar{S},\, \bar{T},\, \bar{S}',\, \bar{T}'. \quad \mathtt{P}_0(\bar{R}, \bar{T}, \bar{T}') \not\sqsubseteq \mathtt{P}'_0(\bar{R}, \bar{T}, \bar{T}') \;\wedge \\
&\bigwedge_{1 \leq j \leq k} \mathtt{P}_j(\bar{R}, \bar{T}, \bar{T}') = S_j \;\wedge \bigwedge_{1 \leq j \leq h} \mathtt{P}'_j(\bar{R}, \bar{T}, \bar{T}') = S'_j \;\wedge \\
&\bigwedge_{\substack{1 \leq j, j' \leq k \\ i_j = i_{j'}}} S_j \subseteq S_{j'} \rightarrow T_j \subseteq T_{j'} \wedge \bigwedge_{\substack{1 \leq j, j' \leq h \\ i'_j = i'_{j'}}} S'_j \subseteq S'_{j'} \rightarrow T'_j \subseteq T'_{j'} \;\wedge
\end{aligned}$$

$$\bigwedge_{\substack{1\leq j\leq k\\ 1\leq j'\leq h\\ i_j=i'_{j'}}} S_j \subseteq S'_{j'} \rightarrow T_j \subseteq T'_{j'} \wedge \bigwedge_{\substack{1\leq j\leq h\\ 1\leq j'\leq k\\ i'_j=i_{j'}}} S'_j \subseteq S_{j'} \rightarrow T'_j \subseteq T_{j'}. \tag{6.3}$$

It is convenient now to rename the relational variables T_j and $T'_{j'}$, where j ranges over $\{1, ..., k\}$ and j' ranges over $\{1, ..., h\}$, by new relational variables U_i, where i ranges over an appropriate set I of indices isomorphic to $\{1, ..., k\} \uplus \{1, ..., h\}$, and, similarly, replace the queries $\mathtt{P}_j(\bar{R}, \bar{T}, \bar{T}')$ and $\mathtt{P}'_j(\bar{R}, \bar{T}, \bar{T}')$ by new queries $\mathtt{O}_i(\bar{R}, \bar{U})$. Accordingly, the conditions of the form $S_j \subseteq S_{j'} \rightarrow T_j \subseteq T_{j'}$ will be replaced by equivalent conditions of the form $\mathtt{O}_i(\bar{R}, \bar{U}) \subseteq \mathtt{O}_{i'}(\bar{R}, \bar{U}) \rightarrow U_i \subseteq U_{i'}$, where the pair (i, i') is either $(0, 0)$ or an element of an appropriate subset D of $I \times I$.
Now, for every partition $\mathcal{D} = (D_+, D_-)$ of D, we denote by $\Sigma_{\mathcal{D}}$ the set of all positive constraints of the form $U_i \subseteq U_{i'}$, with $(i, i') \in D_+$, and all negative constraints of the form $U_i \not\subseteq U_{i'}$, with $(i, i') \in D_-$. Intuitively, each $\Sigma_{\mathcal{D}}$ is a maximal set of containment relationships between the various instances U_i and $U_{i'}$, for all $(i, i') \in D$. Therefore, Property (6.3) holds iff there exists a partition $\mathcal{D} = (D_+, D_-)$ of D such that

$$\exists\, \bar{R}, \bar{U} \vDash \Sigma_{\mathcal{D}}.\ \ \mathtt{O}_0(\bar{R}, \bar{U}) \not\subseteq \mathtt{O}'_0(\bar{R}, \bar{U}) \ \ \wedge \bigwedge_{(i,i')\in D_-} \mathtt{O}_i(\bar{R}, \bar{U}) \not\subseteq \mathtt{O}_{i'}(\bar{R}, \bar{U}). \tag{6.4}$$

We observe that any containment relationship of the form $\mathtt{O}_i(\bar{R}, \bar{U}) \not\subseteq \mathtt{O}'_{i'}(\bar{R}, \bar{U})$, where $\mathtt{O}_i$ is an RA^+-query, is equivalent to an existential quantification over all containment relationships of the form $\tilde{\mathtt{O}}_{i,l}(\bar{R}, \bar{U}) \not\subseteq \mathtt{O}'_{i'}(\bar{R}, \bar{U})$, where $\tilde{\mathtt{O}}_{i,l}$ is a conjunct of the flattening of $\mathtt{O}_i$. This shows that Property (6.4) above is *violated* (and hence $\Phi \subseteq_{\lambda\mathsf{RA}^+} \Phi'$) iff, for every partition $\mathcal{D} = (D_+, D_-)$ of D and every choice of a conjunct $\tilde{\mathtt{O}}_{0,l_0}$ from the flattening of $\mathtt{O}_0$ and for each choice of a conjunct $\tilde{\mathtt{O}}_{i,l_{i,i'}}$ from the flattening of $\mathtt{O}_i$, for each $(i, i') \in D_-$, the following instance of the constrained disjunctive containment problem is satisfied:

$$\forall\, \bar{R}, \bar{U} \vDash \Sigma_{\mathcal{D}}.\ \ \tilde{\mathtt{O}}_{0,l_0}(\bar{R}, \bar{U}) \subseteq \mathtt{O}'_0(\bar{R}, \bar{U}) \ \ \vee \bigvee_{(i,i')\in D_-} \tilde{\mathtt{O}}_{i,l_{i,i'}}(\bar{R}, \bar{U}) \subseteq \mathtt{O}_{i'}(\bar{R}, \bar{U}). \tag{6.5}$$

Such a characterization, together with Lemma 5 (which proves that a conjunct of the flattenings of an RA^+-query can be guessed non-deterministically in polynomial time) and Proposition 6.9 (which proves the NP membership for the constrained disjunctive problem with lefthandside CQs and righthandside terms RA^+-queries, under positive and negative containment constraints), shows that the problem of deciding $\Phi \subseteq_{\lambda\mathsf{RA}^+} \Phi'$ is in Π_2^{P}. □

Note that the following proposition gives immediately a Π_2^{P}-hardness result also for the higher order containment problem $\Phi \subseteq_{\lambda\mathsf{RA}^+} \Phi'$.

Proposition 6.12. *The problem of deciding the containment* $\mathtt{Q} \subseteq \mathtt{Q}'$*, where* $\mathtt{Q}$ *is an* RA^+*-query (indeed, a* $\mathsf{CQ_C}$*) and* $\mathtt{Q}'$ *is a CQ, is* Π_2^{P}*-hard.*

Proof. The proof of this proposition uses the same technique as the Π_2^{P}-hardness proof for the problem of deciding containment between two monotonic relational expressions, see, for instance, [SY80]. The above hardness result, however, strongly relies on the use of constants.

We reduce from the $\forall\exists$-3CNF problem, which has been recalled in Section 3.2, to this containment problem.
We fix two tuples of boolean variables $\bar{x} = (x_1, ..., x_m)$ and $\bar{y} = (y_1, ..., y_n)$ and a 3CNF formula $\bar{\alpha} = \alpha_1 \wedge ... \wedge \alpha_p$ over the variables $\bar{x}, \bar{y}$ and we reduce the problem of deciding $\forall\, \theta_{\bar{x}}\ \exists\, \theta_{\bar{y}}\ (\theta_{\bar{x}}\theta_{\bar{y}} \vDash \bar{\alpha})$ to a containment problem. To do that, we introduce a relational type $\mathcal{R}_i = \big(v_i^1, v_i^2, v_i^3\big)$ for each clause α_i, where the v_i^j's are pairwise distinct attributes with $Dom(v_i^j) = \mathbb{B}$. Any tuple $t \in Dom(\mathcal{R}_i)$, with $1 \leq i \leq p$, represents a possible assignment for the three variables that occur in the clause α_i (therefore, the tuple t encodes the truth value of the clause α_i as well). Accordingly, every relation $\mathtt{R}_i$ of type $\mathcal{R}_i$ represents a set of possible assignments for the three variables that occur in the clause α_i. Clearly, given a tuple of relations $\mathtt{R}_1, ..., \mathtt{R}_p$, their cartesian product $\mathtt{R}_1 \bowtie ... \bowtie \mathtt{R}_p$ may give rise to some inconsistent tuples due to the possible repeated occurrences of the same variable in the various clauses. We get rid of these spurious tuples by applying a selection operator $\sigma_{c(\bar{v}_1,...,\bar{v}_p)}$, where the condition $c(\bar{v}_1, ..., \bar{v}_p)$ enforces the same truth value on two attributes $v_i^j, v_{i'}^{j'}$ whenever the variables in the corresponding clauses $\alpha_i^j, \alpha_{i'}^{j'}$ are the same, namely:

$$c(\bar{v}_1, ..., \bar{v}_p) \;=\; \bigwedge_{\substack{1\leq i,i'\leq p \\ 1\leq j,j'\leq 3 \\ \alpha_i^j=\alpha_{i'}^{j'} \vee \alpha_i^j=\neg\alpha_{i'}^{j'}}} v_i^j = v_{i'}^{j'}.$$

We can now define the lefthandside term $\mathtt{Q}$ and the righthandside term $\mathtt{Q}'$ for the corresponding instance of the containment problem. For each clause α_i, we list the seven (out of eight) assignments for $\bar{v}_i = (v_i^1, v_i^2, v_i^3)$ that satisfy α_i. Let $t_{i,1}, ..., t_{i,7}$ be these assignments. We then define the $\mathsf{CQ_C}$

$$\mathtt{Q} \;=\; \lambda R_1 ... \lambda R_p.\ \bigwedge_{\substack{1\leq i\leq p \\ 1\leq j\leq 7}} \pi_{\emptyset}\big(\sigma_{\bar{v}_i=t_{i,j}}(R_i)\big) \;\bowtie\; \pi_X\big(\sigma_{c(\bar{v}_1,...,\bar{v}_p)}(\mathtt{C}_1 \bowtie ... \bowtie \mathtt{C}_p)\big)$$

where each $\mathtt{C}_i$ is a relational constant of type $\mathcal{R}_i$ that contains the eight possible assignments for $\bar{v}_i = (v_i^1, v_i^2, v_i^3)$ and X is the set of all attributes v_i^j for which the corresponding literal α_i^j belongs to $\{x_1, ..., x_n, \neg x_1, ..., \neg x_n\}$. Intuitively, the CQ_Cterm $\mathtt{Q}$ above receives as input a tuple of relations $\mathtt{R}_1, ..., \mathtt{R}_p$ and returns either the set of all possible (consistent) assignments for the occurrences of the variables $\{x_1, ..., x_m\}$ in the clauses $\alpha_1, ..., \alpha_p$ or the empty set, depending on whether or not, for every $1 \leq i \leq p$, $\mathtt{R}_i$ contains all tuples among $t_{i,1}, ..., t_{i,7}$ (namely, all possible ways of satisfying the clause α_i). As for the righthandside term, we simply define the CQ

$$\mathtt{Q}' \;=\; \lambda R_1 ... \lambda R_p.\;\; \pi_X\big(\sigma_{c(\bar{v}_1,...,\bar{v}_p)}(R_1 \bowtie ... \bowtie R_p)\big).$$

The output of $\mathtt{Q}'$ on some relations $\mathtt{R}_1, ..., \mathtt{R}_p$ is the set of assignments for the occurrences of the variables $\bar{x}$ obtained by projecting the consistent assignments from $\mathtt{R}_1 \bowtie ... \bowtie \mathtt{R}_p$.
We now prove that

$$\mathtt{Q} \subseteq \mathtt{Q}' \qquad \text{iff} \qquad \forall\, \theta_{\bar{x}}\; \exists\, \theta_{\bar{y}}\;\; \big(\theta_{\bar{x}}\theta_{\bar{y}} \vDash \bar{\alpha}\big).$$

$\Rightarrow$) Assume that $\mathtt{Q} \subseteq \mathtt{Q}'$ and let T be the set of all assignments θ for $\bar{x}$ and $\bar{y}$ that satisfy $\bar{\alpha}$. For every $1 \leq i \leq p$, we let $\mathtt{R}_i$ be the relation that consists of all tuples t for which there is $\theta \in T$ such that $t.v_i^j$ coincides with the value of the variable in α_i^j under the assignment θ. Clearly, $\pi_\emptyset\big(\sigma_{\bar{v}_i = t_{i,j}}(\mathtt{R}_i)\big)$ evaluates to $\mathtt{true}$, for all $1 \leq i \leq p$ and all $1 \leq j \leq 7$, and hence $\mathtt{Q}(\mathtt{R}_1, ..., \mathtt{R}_p)$ is the set X of all possible consistent assignments for the occurrences of the variables $\{x_1, ..., x_m\}$ in the clauses $\alpha_1, ..., \alpha_p$. Since $\mathtt{Q} \subseteq \mathtt{Q}'$, we know that $\mathtt{Q}'(\mathtt{R}_1, ..., \mathtt{R}_p)$ contains at least all tuples from X, which basically shows that every assignment $\theta_{\bar{x}}$ for $\bar{x}$ can be extended to an assignment θ for $\bar{x}$ and $\bar{y}$ satisfying $\bar{\alpha}$.

$\Leftarrow$) Suppose that every assignment $\theta_{\bar{x}}$ for $\bar{x}$ can be extended to an assignment θ for $\bar{x}$ and $\bar{y}$ satisfying $\bar{\alpha}$. Fix some generic relations $\mathtt{R}_1, ..., \mathtt{R}_p$ and assume, without loss of generality, that $\mathtt{Q}(\mathtt{R}_1, ..., \mathtt{R}_p)$ is a *non-empty* set X (if this were not the case, then the inclusion $\mathtt{Q}(\mathtt{R}_1, ..., \mathtt{R}_p) \subseteq \mathtt{Q}'(\mathtt{R}_1, ..., \mathtt{R}_p)$ would follow trivially). Let t be a generic tuple in X. By construction, t represents an assignment $\theta_{\bar{x}}^t$ for the occurrences of the variables $\bar{x}$ in the clauses $\alpha_1, ..., \alpha_p$, which is obtained by projecting some generic (consistent) assignment from $\mathtt{C}_1 \bowtie ... \bowtie \mathtt{C}_p$. From the hypothesis, we know that such an assignment $\theta_{\bar{x}}^t$ can be extended to an assignment θ^t for the occurrences of the variables $\bar{x}$ and $\bar{y}$ in such a way that all clauses $\alpha_1, ..., \alpha_p$ are satisfied simultaneously. Finally, since $X \neq \emptyset$, we have

that such an extension θ^t belongs to the product $\mathtt{R}_1 \bowtie ... \bowtie \mathtt{R}_p$ and hence $X \subseteq \mathtt{Q}'(\mathtt{R}_1, ..., \mathtt{R}_p)$.

This completes the proof of the claim. □

Proposition 6.11 and Proposition 6.12 together give precisely the claim of Theorem 6.8. Moreover, in the proof of Proposition 6.11, we use only a few main properties, in particular: (i) the constrained disjunctive containment for lefthandside CQs and righthandside RA^+-queries, under positive and negative containment constraints, is in NP, and (ii) the set of all possible queries that can be used to instantiate an order 1 variable is as expressive as the set of all monotone queries. Therefore, we can extend the result as follows:

Corollary 6.13. *Let $\mathsf{RA}^{+,\neq}$ be the signature that extends RA^+ with selection operators that use equalities and inequalities between attributes, or between attributes and constants. Then, the problem of deciding the containment $\Phi \subseteq_{\lambda\mathsf{RA}^{+,\neq}} \Phi'$, where $\Phi, \Phi' \in \mathsf{HO}_2^{\downarrow}[\mathsf{RA}^+]$, is Π_2^{P}-complete.*

Adding dependencies.

We now consider higher-order containment relative to *integrity constraints*. We focus on two widely-studied constraint classes, namely, functional dependencies and inclusion dependencies [AHV95]. The containment problem for CQs under sets of functional dependencies has been deeply investigated starting from [ASU79a] and it is known to be NP-complete.

Below, given two higher-order queries $\Phi, \Phi' \in \mathsf{HO}_2^{\downarrow}[\mathsf{RA}^+]$ of the same type and given a set Δ of constraints (e.g., functional dependencies) over the formal arguments of Φ and Φ', we write $\Phi \subseteq_{\lambda\mathsf{RA}^+,\Delta} \Phi'$ iff, for every input $\bar{\mathtt{Q}}, \bar{\mathtt{R}}$ that satisfies the constraints in Δ, we have $[\![\Phi]\!]_{\lambda\mathsf{RA}^+}(\bar{\mathtt{Q}}, \bar{\mathtt{R}}) \subseteq [\![\Phi']\!]_{\lambda\mathsf{RA}^+}(\bar{\mathtt{Q}}, \bar{\mathtt{R}})$.

We can extend Theorem 6.8 to this setting:

Theorem 6.14. *The problem of deciding the containment $\Phi \subseteq_{\lambda\mathsf{RA}^+,\Delta} \Phi'$, where $\Phi, \Phi' \in \mathsf{HO}_2^{\downarrow}[\mathsf{RA}^+]$ and Δ is a set of functional dependencies, is Π_2^{P}-complete.*

The proof of the complexity upper bound goes along the same lines of the proof of Proposition 6.11. More precisely, we first exploit Proposition 6.10 (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 lefthandside CQs

and righthandside RA^+-queries, under positive and negative containment constraints and the additional functional dependencies. We then argue that the latter variant of the disjunctive containment problem is in NP:

Proposition 6.15. *The disjunctive containment problem for lefthandside CQs and righthandside* RA^+*-queries, under positive and negative containment constraints and functional dependencies, is NP-complete.*

Proof. By following the same reduction of the proof of Proposition 6.9, we first claim:

Lemma 6. *The disjunctive containment problem for lefthandside CQs and righthandside* RA^+*-queries, under positive and negative containment constraints and functional dependencies, can be reduced in polynomial time to the containment problem for lefthandside CQs and righthandside* RA^+*-queries, under positive containment constraints and functional dependencies.*

The proof of the above result is almost the same as that of Lemma 3 and thus omitted. The following result is also similar to Lemma 4.

Lemma 7. *The containment problem for lefthandside CQs and righthandside* RA^+*-queries, under positive containment constraints and functional dependencies, is in NP.*

Proof. We prove the claim for instances involving boolean queries. Let $\mathtt{B}$ be a boolean CQ of type $(\mathcal{R}_1 \times ... \times \mathcal{R}_n) \to \{\}$, let $\mathtt{B}'$ be a boolean RA^+-query of the same type, and let Σ and Δ be, respectively, a set of positive containment constraints and a set of functional dependencies, over objects of type $\bar{\mathcal{R}} = \mathcal{R}_1 \times ... \times \mathcal{R}_n$. We transform $\mathtt{B}$ into a new query $\tilde{\mathtt{B}}$ by applying a variant of the chase procedure for inclusion dependencies and functional dependencies [AHV95, JK84]. Formally, $\tilde{\mathtt{B}}$ is obtained by applying exhaustively first Rule 1. and then Rule 2. below:

1. Add a new conjunct of the form $R_j(t)$, where t is a tuple mapping the attributes of $\mathcal{R}_i$ to some variables and constants, whenever $R_i \subseteq R_j$ is a containment constraint in Σ and $R_i(t)$ appears as a conjunct in the current expansion of $\mathtt{B}$.

2. For every pair of tuples t, t', mapping attributes of $\mathcal{R}_i$ to variables and constants, identify (by using either equalities or substitutions) any two variables/constants $t.b$ and $t'.b$ associated with the same attribute $b \in B$ whenever $R_i.A \to R_i.B$ is a functional dependency in Δ and the current expansion of $\mathtt{B}$ contains two conjuncts of the form $R_i(t)$ and $R_i(t')$, with $\pi_A(t) = \pi_A(t')$.

Clearly, both the expansion $\tilde{\mathtt{B}}$ and its canonical model $\bar{\mathtt{R}}^{\tilde{\mathtt{B}}}$ have size polynomial in the size of $\mathtt{B}$. We now claim:

$$\mathtt{B} \subseteq_{\Sigma,\Delta} \mathtt{B}' \quad \text{iff} \quad \mathtt{B}'(\bar{\mathtt{R}}^{\tilde{\mathtt{B}}}) = \mathtt{true}.$$

$\Rightarrow$) Assume that $\mathtt{B} \subseteq_{\Sigma,\Delta} \mathtt{B}'$. By definition, the canonical model $\bar{\mathtt{R}}^{\tilde{\mathtt{B}}}$ satisfies $\tilde{\mathtt{B}}$, and hence $\mathtt{B}$ as well. Given the definition of $\tilde{\mathtt{B}}$, we have that $\bar{\mathtt{R}}^{\tilde{\mathtt{B}}}$ satisfies also the containment constraints in Σ and the functional dependency in Δ. Thus, knowing that $\mathtt{B} \subseteq_{\Sigma,\Delta} \mathtt{B}'$, we immediately derive $\mathtt{B}'(\bar{\mathtt{R}}^{\tilde{\mathtt{B}}}) = \mathtt{true}$.

$\Leftarrow$) Assume that $\mathtt{B}'(\bar{\mathtt{R}}^{\tilde{\mathtt{B}}}) = \mathtt{true}$. From [CM77] (see also Equation 6.1 in the proof of Lemma 4), we know that there is a homomorphism h' from the canonical model $\bar{\mathtt{R}}^{\mathtt{B}'}$ of $\mathtt{B}'$ to the canonical model $\bar{\mathtt{R}}^{\tilde{\mathtt{B}}}$ of $\tilde{\mathtt{B}}$. Let $\bar{\mathtt{R}}$ be a tuple of relations that satisfy the query $\mathtt{B}$, the containment constraints in Σ, and the functional dependencies in Δ. By construction, we have $\tilde{\mathtt{B}}(\bar{\mathtt{R}}) = \mathtt{true}$. Again from [CM77], we know that there is another homomorphism h from $\bar{\mathtt{R}}^{\tilde{\mathtt{B}}}$ to $\bar{\mathtt{R}}$. The functional composition $h \circ h'$ is a homomorphism from $\bar{\mathtt{R}}^{\mathtt{B}'}$ to $\bar{\mathtt{R}}$. Therefore, by applying once more the characterization from [CM77], we conclude $\mathtt{B}'(\bar{\mathtt{R}}) = \mathtt{true}$.

In order to decide whether $\mathtt{B} \subseteq_{\Sigma,\Delta} \mathtt{B}'$ it is sufficient to guess (i) a conjunct $\tilde{\mathtt{B}}'_i$ of the flattening of $\mathtt{B}'$ (see Lemma 5) and (ii) a homomorphism from $\bar{\mathtt{R}}^{\tilde{\mathtt{B}}'_i}$ to $\bar{\mathtt{R}}^{\tilde{\mathtt{B}}}$ witnessing $\mathtt{B}'(\bar{\mathtt{R}}^{\tilde{\mathtt{B}}}) = \mathtt{true}$. □

The proof of Proposition 6.15 is, not surprisingly, analogous to that of Proposition 6.9 and thus omitted. □

Now, we turn towards higher-order containment in the setting of inclusion dependencies.

Theorem 6.16. *The problem of deciding the containment* $\Phi \subseteq_{\lambda\mathsf{RA}^+,\Delta} \Phi'$*, where* $\Phi, \Phi' \in \mathsf{HO}_2^{\downarrow}[\mathsf{RA}^+]$ *and* Δ *is a set of inclusion dependencies, is PSPACE-complete.*

Proof. It is known that the containment problem between two CQs under a set Δ of inclusion dependencies is PSPACE-hard (see, for instance, [CFP84]). In addition, CQs, considered as constant functionals, are special cases of higher-order queries over the signature CQ. Thus, the higher order containment problem under a set of inclusion dependencies is PSPACE-hard as well.
We now prove the PSPACE upper bound. Using the same transformation as in the proof of Theorem 6.8, we reduce the higher order containment problem under a set Δ

of inclusion dependencies to the problem of universally guessing and deciding suitable instances of the disjunctive containment problem that have the following form:

$$\forall\, \bar{R}, \bar{U} \vDash \Sigma_{\mathcal{D}} \cup \Delta.\ \ \tilde{\mathtt{O}}_{0,l_0}(\bar{R}, \bar{U}) \subseteq \mathtt{O}'_0(\bar{R}, \bar{U}) \ \ \vee \bigvee_{(i,i') \in D_-} \tilde{\mathtt{O}}_{i,l_{i,i'}}(\bar{R}, \bar{U}) \subseteq \mathtt{O}_{i'}(\bar{R}, \bar{U}).$$

where $\Sigma_{\mathcal{D}}$ is a set of positive and negative containment constraints and Δ is the set of inclusion dependencies.
Now, we observe that positive containment constraints are special forms of inclusion dependencies. Thus, in order to decide the above property, it is sufficient to consider the disjunctive containment problem for lefthandside CQs and righthandside RA^+-queries, under negative containment constraints and inclusion dependencies. By a straightforward generalization of the proof of Proposition 6.9, this problem can be reduced to the containment problem for lefthandside CQs and righthandside RA^+-queries, under inclusion dependencies only. Finally, the latter problem can be solved in polynomial space by guessing a conjunct of the flattening of the righthandside RA^+-query and by deciding a classical containment problem between CQs under inclusion dependencies, which is known to be in PSPACE [JK84]. □

6.3.2 Tractable cases

We conclude this section by considering special instances of the higher-order containment problem that can be solved efficiently, namely, by a non-deterministic polynomial-time algorithm (or, even better, by a deterministic polynomial-time algorithm).

Definition 13. *We define the class of* single-argument terms *as the least set that contains all terms of the form:*

- $\mathtt{Q}(R_1, ..., R_n)$*, where* $R_1, ..., R_n$ *are relational variables and* $\mathtt{Q}$ *is an* RA^+*-query with* n *formal arguments;*
- $\mathtt{Q}\big(Q(\tau), ..., Q(\tau)\big)$*, where* τ *is a single-argument term with at most one free query variable* Q *and* $\mathtt{Q}$ *is an* RA^+*-query, whose input is instantiated with as many copies of the term* $Q(\tau)$ *as the number of formal arguments of* $\mathtt{Q}$*.*

We then define single-argument higher-order queries *as the closures (by* λ*-abstraction over all free variables) of single-argument terms.*

We associate with each single-argument higher-order query Φ the (unique) sequence of RA^+-queries that generates the body of Φ in the grammar above, namely, the sequence $\mathtt{Q}_1, ..., \mathtt{Q}_n$ such that $\Phi = \lambda Q.\, \lambda \bar{R}.\, \mathtt{Q}_n\big(..., Q(\mathtt{Q}_{n-1}(...)), ..\big)$. We call this sequence the *generating sequence* for τ and its length the *nesting-depth* of Φ.

Example 16. *The term* $\lambda Q.\, \lambda R.\, \rho_{a/b}\big(Q(R)\big) \bowtie \rho_{a'/b'}\big(Q(R)\big)$ *is a single-argument higher-order query, whose generating sequence consists of single* RA^+*-query* $\mathtt{Q}_1 = \lambda S.\, \rho_{a/b}(S) \bowtie \rho_{a'/b'}(S)$*. On the other hand, the term* $\lambda Q.\, \lambda R_1.\, \lambda R_2.\, Q(R_1) \bowtie Q(R_2)$ *is not a single-argument higher-order query, since the two formal arguments of the operator* $\bowtie$ *are instantiated with syntactically different terms.*

Hereafter, we say that a query $\mathtt{Q}$ is *non-constant* if its equivalent rule-based form has at least one variable in the head. In the special case of single-argument higher-order queries where the generating sequences consists of non-constant RA^+-queries only, we can reduce higher-order containment to ordinary containment:

Proposition 6.17. *Given two single-argument higher-order queries* Φ, Φ' *of the same type and with generating sequences* $\mathtt{Q}_1, ..., \mathtt{Q}_m$ *and* $\mathtt{Q}'_1, ..., \mathtt{Q}'_n$*, both consisting of non-constant* RA^+*-queries, we have*

$$\Phi \subseteq_{\lambda\mathsf{RA}^+} \Phi' \quad \textit{iff} \quad \begin{cases} m = n \\ \mathtt{Q}_i \subseteq \mathtt{Q}'_i \quad \textit{for all } 1 \leq i \leq m. \end{cases}$$

Proof. The "if" direction is trivial. We thus prove the opposite direction. Let Q be the unique query variable that appear in the higher-order queries Φ and Φ' (up to a renaming, we can assume that the variable is the same), let $\mathtt{Q}_1, ..., \mathtt{Q}_m$ be the generating sequence for Φ and let $\mathtt{Q}'_1, ..., \mathtt{Q}'_m$ be the generating sequence for Φ'. We assume that either $m \neq n$ (case 1. below), or $(m = n)$ and $\mathtt{Q}_i \not\subseteq \mathtt{Q}'_i$ for some $1 \leq i \leq n$ (case 2. below), and we prove that $\Phi \not\subseteq_{\lambda\mathsf{RA}^+} \Phi'$ as follows.

1. Suppose that $m \neq n$. We will build instances for the query variable Q and for the relational variables $\bar{R}$ in such a way that the containment of Φ in Φ' is violated. We choose an arbitrary tuple $\bar{\mathtt{R}}_0$ of relations to be provided as input to the higher-order queries Φ and Φ'. Since all queries $\mathtt{Q}_i$ and $\mathtt{Q}'_i$ contain free variables in the head, we can provide each $\mathtt{Q}_i$ and each $\mathtt{Q}'_i$ with different inputs in such a way that the corresponding outputs are not contained in each other. This gives rise to a sequence $\bar{\mathtt{R}}_1, ..., \bar{\mathtt{R}}_{\max(m,n)}$ of relational instances of appropriate types that satisfies (i) $\mathtt{Q}_i(\bar{\mathtt{R}}_i) \not\subseteq \mathtt{Q}_j(\bar{\mathtt{R}}_j)$, for all pair of distinct indices $1 \leq i, j \leq m$, (ii) $\mathtt{Q}'_i(\bar{\mathtt{R}}_i) \not\subseteq \mathtt{Q}'_j(\bar{\mathtt{R}}_j)$, for all pair of distinct indices $1 \leq i, j \leq n$,

and (iii) $\mathtt{Q}_i(\bar{\mathtt{R}}_i) = \mathtt{Q}'_i(\bar{\mathtt{R}}_j)$ for all pair of distinct indices $1 \leq i, j \leq \min(m, n)$. Moreover, we can instantiate the query variable Q with a suitable RA^+-query $\mathtt{Q}$ in such a way that $\mathtt{Q}\big(\mathtt{Q}_i(\bar{\mathtt{R}}_i)\big) = \bar{\mathtt{R}}_{i+1}$, for all $1 \leq i < m$, and $\mathtt{Q}\big(\mathtt{Q}'_i(\bar{\mathtt{R}}_i)\big) = \bar{\mathtt{R}}_{i+1}$, for all $1 \leq i < n$. With these assignments, the term $\Phi(\mathtt{Q}, \bar{\mathtt{R}}_0)$ will evaluate to $\mathtt{Q}_m(\bar{\mathtt{R}}_m)$ and, similarly, the term $\Phi'(\mathtt{Q}, \bar{\mathtt{R}}_0)$ will evaluate to $\mathtt{Q}'_n(\bar{\mathtt{R}}_n)$. Finally, since $\mathtt{Q}_m(\bar{\mathtt{R}}_m) \not\subseteq \mathtt{Q}'_n(\bar{\mathtt{R}}_n)$, we conclude that $\Phi \not\subseteq_{\lambda\mathsf{RA}^+} \Phi'$.

2. Suppose that $m = n$ and $\mathtt{Q}_i \not\subseteq \mathtt{Q}'_i$ for some index $1 \leq i \leq m$. Let k be the smallest index such that $\mathtt{Q}_k \not\subseteq \mathtt{Q}'_k$. Clearly, for all $1 \leq i < k$, we have $\mathtt{Q}_i \subseteq \mathtt{Q}'_i$. Since $\mathtt{Q}_k \not\subseteq \mathtt{Q}'_k$, there is a relational constant $\bar{\mathtt{R}}_0$ such that $\tau_k(\bar{\mathtt{R}}_0) \not\subseteq \tau'_k(\bar{\mathtt{R}}_0)$. As before, by exploiting the fact that all queries τ_i and τ'_i are non-constant, we can find two sequences of instances $\bar{\mathtt{R}}^0_0, ..., \bar{\mathtt{R}}^0_m$ and $\bar{\mathtt{R}}'_0, ..., \bar{\mathtt{R}}'_n$ such that (i) $\bar{\mathtt{R}}^0_i = \bar{\mathtt{R}}'_i$, for all $1 \leq i \leq k$, (ii) $\tau_i(\bar{\mathtt{R}}^0_i) \not\subseteq \tau_j(\bar{\mathtt{R}}^0_j)$ and $\tau'_i(\bar{\mathtt{R}}'_i) \not\subseteq \tau'_j(\bar{\mathtt{R}}'_j)$, for all pairs of distinct indices $1 \leq i, j \leq m$, and (iii) $\tau_i(\bar{\mathtt{R}}^0_i) \not\subseteq \tau'_j(\bar{\mathtt{R}}'_j)$, for all $i, j \geq k$. We finally let $\mathtt{Q}$ be a RA^+-query such that $\mathtt{Q}\big(\mathtt{Q}_{i-1}(\bar{\mathtt{R}}^0_{i-1})\big) = \bar{\mathtt{R}}^0_i$ and $\mathtt{Q}\big(\mathtt{Q}'_{i-1}(\bar{\mathtt{R}}'_{i-1})\big) = \bar{\mathtt{R}}'_i$. Since $\mathtt{Q}_m(\bar{\mathtt{R}}^0_m) \not\subseteq \tau'_n(\bar{\mathtt{R}}'_n)$, we conclude that $\Phi \not\subseteq_{\lambda\mathsf{RA}^+} \Phi'$.

The results from the two cases above lead to the proof of the proposition. □

From Proposition 6.17, we immediately obtain the following result:

Theorem 6.18. *The problem of deciding the containment $\Phi \subseteq_{\lambda\mathsf{RA}^+} \Phi'$, where Φ, Φ' are single-argument higher-order queries, with generating sequences consisting of non-constant UCQs, is NP-complete.*

Moreover, if we further restrict the single-argument higher-order queries in such a way that their generating sequences contain only non-constant queries in a certain tractable class, then we immediately obtain an analogous class of higher-order queries for which the containment problem turns out to be tractable (i.e., in P). For instance, consider the case of *acyclic* CQs, where evaluation becomes tractable [Yan81]. Likewise, we have that containment of UCQs in acyclic CQs is tractable. We can then extend this to:

Corollary 6.19. *The problem of deciding the containment $\Phi \subseteq_{\lambda\mathsf{RA}^+} \Phi'$, where Φ is a single-argument higher-order query, with generating sequence consisting of non-constant UCQs, and Φ' is a single-argument higher-order query, with generating sequence consisting of non-constant acyclic CQs, is in P time.*

We can easily replace, in the above result, the acyclicity condition over order 1 queries by other conditions that guarantee tractability for ordinary conjunctive query containment (e.g., bounded treewidth, bounded hyper-treewidth [GLS02]).

6.3.3 Higher-order containment in full Relational Algebra base

In this subsection we focus on the higher-order containment problem for the case where query variables are instantiated by queries of the full Relational Algebra. We will still restrict the constant operators used in the higher-order queries to range over the signature RA^{+}, since it is well-known that the containment problem for terms built up from the full Relational Algebra is undecidable. In contrast to this, we show that extending the base does *not* make higher-order containment harder.

Theorem 6.20. *The problem of deciding the containment* $\Phi \subseteq_{\lambda\mathsf{RA}} \Phi'$*, where* $\Phi, \Phi' \in \mathsf{HO}_2^{\downarrow}[\mathsf{RA}^{+}]$*, is* Π_2^{P}*-complete.*

Proof. The complexity lower bound is trivial from previous results. As regards the complexity upper bound, we remark here that the key ingredient, as before, is a "quantifier elimination" property, namely, the analog of Proposition 6.10 for queries quantified over the full Relational Algebra:

Proposition 6.21. *Fix* $m > 0$ *and, for all* $1 \leq i \leq m$*, let* $\bar{\mathcal{S}} \to \mathcal{T}_i$ *be an order* 1 *query type. Moreover, fix* $k > 0$ *and, for all* $1 \leq j \leq k$*, let (i)* i_j *be an index from* $\{1, ..., m\}$*, (ii)* $\bar{\mathtt{S}}_j$ *be a tuple of relations of types in* $\bar{\mathcal{S}}$*, and (iii)* $\mathtt{T}_j$ *be a relation of type* $\mathcal{T}_{i_j}$*. The following properties are equivalent:*

1. *there exist some* RA*-queries* $\mathtt{Q}_1, ..., \mathtt{Q}_m$ *such that* $\mathtt{Q}_{i_j}(\bar{\mathtt{S}}_j) = \mathtt{T}_j$ *for all* $j \in \{1, ..., k\}$*;*
2. *for every pair of indices* $j, j' \in \{1, ..., k\}$*, if* $i_j = i_{j'}$ *and* $\bar{\mathtt{S}}_j = \bar{\mathtt{S}}_{j'}$*, then* $\mathtt{T}_j = \mathtt{T}_{j'}$*.*

Along the same lines of the proof of Theorem 6.8, we first associate new relations with the subterms of Φ and Φ' and we then exploit Proposition 6.21 in order to get rid of the existential quantification over the query variables. The analogue of Property 6.3 in Theorem 6.8 is as follows:

$$
\begin{aligned}
\exists\, \bar{R},\, \bar{S},\, \bar{T},\, \bar{S}',\, \bar{T}'. \quad & \mathtt{P}_0(\bar{R}, \bar{T}, \bar{T}') \not\subseteq \mathtt{P}'_0(\bar{R}, \bar{T}, \bar{T}') \quad \wedge \\
& \bigwedge_{1 \leq j \leq k} \mathtt{P}_j(\bar{R}, \bar{T}, \bar{T}') = S_j \quad \wedge \quad \bigwedge_{1 \leq j \leq h} \mathtt{P}'_j(\bar{R}, \bar{T}, \bar{T}') = S'_j \quad \wedge \\
& \bigwedge_{\substack{1 \leq j, j' \leq k \\ i_j = i_{j'}}} S_j = S_{j'} \to T_j = T_{j'} \quad \wedge \quad \bigwedge_{\substack{1 \leq j, j' \leq h \\ i'_j = i'_{j'}}} S'_j = S'_{j'} \to T'_j = T'_{j'} \quad \wedge \\
& \bigwedge_{\substack{1 \leq j \leq k \\ 1 \leq j' \leq h \\ i_j = i'_{j'}}} S_j = S'_{j'} \to T_j = T'_{j'} \quad \wedge \quad \bigwedge_{\substack{1 \leq j \leq h \\ 1 \leq j' \leq k \\ i'_j = i_{j'}}} S'_j = S_{j'} \to T'_j = T_{j'}.
\end{aligned}
$$

Further transformations would then give the following characterization of the higher order containment problem. Given Φ and Φ', we can compute a set D of pairs of indices and some RA^+-queries $\mathtt{O}_0, \mathtt{O}'_0, \mathtt{O}_i, \mathtt{O}'_i$ such that $\Phi \subseteq_{\lambda\mathsf{RA}} \Phi'$ holds iff, for every partition $\mathcal{D} = (D_=, D_{\neq})$ of D and every choice of a conjunct $\tilde{\mathtt{O}}_{0,l_0}$ from the flattening of $\mathtt{O}_0$ and some conjuncts $\tilde{\mathtt{O}}_{i,l_{i,i'}}$ from the flattening of $\mathtt{O}_i$, for each $(i,i') \in D_{\neq}$, the following instance of the constrained disjunctive containment problem is satisfied:

$$\forall\, \bar{R}, \bar{U} \vDash \Sigma_{\mathcal{D}}. \qquad \tilde{\mathtt{O}}_{0,l_0}(\bar{R}, \bar{U}) \subseteq \mathtt{O}'_0(\bar{R}, \bar{U}) \quad \vee \bigvee_{(i,i') \in D_{\neq}} \tilde{\mathtt{O}}_{i,l_{i,i'}}(\bar{R}, \bar{U}) = \mathtt{O}_{i'}(\bar{R}, \bar{U}).$$

Finally, by rewriting equality (resp., inequality) constraints as conjunctions (resp., disjunctions) of containment (resp., non-containment) relationships, and by exploiting, as usual, Proposition 6.9 and Lemma 5, we can conclude that the higher order containment problem in the RA-base is still in Π^{P}_2. □

6.3.4 Higher-order containment in CQ base

We now consider the situation when we move to the case that the union operator is removed. Here we show that moving to the conjunctive query base does not make the higher-order containment problem easier. Indeed, Proposition 6.12 gives immediately the following hardness result:

Corollary 6.22. *Let $\mathcal{I}$ be any arbitrary interpretation for the query variables (e.g., $\mathcal{I} = \lambda\mathsf{CQ}_{\mathsf{C}}$). The problem of deciding the containment $\Phi \subseteq_{\mathcal{I}} \Phi'$, where $\Phi, \Phi' \in \mathsf{HO}^{\downarrow}_2[\mathsf{CQ}_{\mathsf{C}}]$ is Π^{P}_2-hard. The lower bound holds also in the case where Φ or Φ', or both of them, contains no occurrences of query variables.*

A similar lower bound holds for the higher-order containment problem in the signature CQ:

Proposition 6.23. *The problem of deciding the containment $\Phi \subseteq_{\lambda\mathsf{CQ}} \Phi'$, where $\Phi, \Phi' \in \mathsf{HO}^{\downarrow}_2[\mathsf{CQ}]$, Φ contains one occurrence of a query variable, and Φ' contains no occurrences of query variables, is Π^{P}_2-hard.*

Proof. Similar to the proof of Proposition 6.12, we reduce from the $\forall\exists$-3CNF problem, which is Π^{P}_2-complete. An instance of the $\forall\exists$-3CNF problem consists of two tuples $\bar{x} = (x_1, ..., x_m)$ and $\bar{y} = (y_1, ..., y_n)$ of boolean variables and a 3CNF formula $\bar{\alpha} = \alpha_1 \wedge ... \wedge \alpha_p$, where each clause α_i is a disjunction of exactly three literals (i.e., variables from $\{x_1, ..., x_m, y_1, ..., y_n\}$ or their negations), denoted α^1_i, α^2_i, and α^3_i. The problem

consists of deciding whether for every assignment $\theta_{\bar{x}}$ for $\bar{x}$, there exists an assignment $\theta_{\bar{y}}$ for $\bar{y}$ that satisfies $\bar{\alpha}$ (shortly, $\theta_{\bar{x}}\theta_{\bar{y}} \vDash \bar{\alpha}$).
We fix two tuples of boolean variables $\bar{x} = (x_1, ..., x_m)$ and $\bar{y} = (y_1, ..., y_n)$ and a 3CNF formula $\bar{\alpha} = \alpha_1 \wedge ... \wedge \alpha_p$ over the variables $\bar{x}, \bar{y}$ and we reduce the problem of deciding $\forall\, \theta_{\bar{x}}\ \exists\, \theta_{\bar{y}}\ (\theta_{\bar{x}}\theta_{\bar{y}} \vDash \bar{\alpha})$ to a containment problem. To do that, we introduce a relational type $\tau_i = \big(v_i^1, v_i^2, v_i^3\big)$ for each clause α_i, where the v_i^j's are pairwise distinct attributes with $Dom(v_i^j) = \mathbb{B}$. Any tuple $t \in Dom(\tau_i)$, with $1 \leq i \leq p$, represents a possible assignment for the three variables that occur in the clause α_i (therefore, the tuple t encodes the truth value of the clause α_i as well). Accordingly, every relation R_i of type τ_i represents a set of possible assignments for the three variables that occur in the clause α_i. Clearly, given a tuple of relations $R_1, ..., R_p$, their cartesian product $R_1 \bowtie ... \bowtie R_p$ may give rise to some inconsistent tuples due to the possible repeated occurrences of the same variable in the various clauses. We get rid of these spurious tuples by applying a selection operator $\sigma_{c(\bar{v}_1,...,\bar{v}_p)}$, where the condition $c(\bar{v}_1, ..., \bar{v}_p)$ enforces the same truth value on two attributes $v_i^j, v_{i'}^{j'}$ whenever the variables in the corresponding clauses $\alpha_i^j, \alpha_{i'}^{j'}$ are the same, namely:

$$c(\bar{v}_1, ..., \bar{v}_p) \;=\; \bigwedge_{\substack{1\leq i,i'\leq p \\ 1\leq j,j'\leq 3 \\ \alpha_i^j=\alpha_{i'}^{j'} \vee \alpha_i^j=\neg\alpha_{i'}^{j'}}} v_i^j = v_{i'}^{j'}.$$

We use a query variable Q which maps a two-attribute relational type to an m-attribute type $\{B_1, \ldots, B_m\}$ with attribute ranges $\mathbb{Z}$. Later we will see that each variable x_i corresponds to an attribute B_i in the output of query Q over a specific relation. We call that relation $R_0[A_0^1, A_0^2]$. We can now define the lefthandside term Φ and the righthandside term Φ' for the corresponding instance of the containment problem. For each clause α_i, we list the seven (out of eight) assignments for $\bar{v}_i = (v_i^1, v_i^2, v_i^3)$ that satisfy α_i. Let $t_{i,1}, ..., t_{i,7}$ be these assignments. We then define

$$\begin{aligned}\Phi \;=\; \lambda Q.\, \lambda R_0 ... \lambda R_p.\ & \bigwedge_{\substack{1\leq i\leq p \\ 1\leq j\leq 7}} \pi_\emptyset\big(\sigma_{\bar{v}_i=t_{i,j}}(R_i)\big) \quad \bowtie \\ & \pi_\emptyset\big(\sigma_{B_i=2}(Q(\sigma_{(A_0^1,A_0^2)=(2,2)}R_0))\big) \quad \bowtie \\ & \pi_\emptyset\big(\sigma_{B_i=3}(Q(\sigma_{(A_0^1,A_0^2)=(3,3)}R_0))\big) \quad \bowtie \\ & Q(\sigma_{(A_0^1,A_0^2)=(0,1)}R_0)\end{aligned}$$

Intuitively, Φ returns $Q(\sigma_{(A_0^1,A_0^2)=(0,1)}R_0)$ iff the relation instance of R_i contains all satisfiable assignments for α_i, together with $\sigma_{B_i=2}(Q(\sigma_{(A_0^1,A_0^2)=(2,2)}R_0)) \neq \emptyset$, $\sigma_{B_i=3}(Q(\sigma_{(A_0^1,A_0^2)=(3,3)}R_0)) \neq \emptyset$, and $Q(\sigma_{(A_0^1,A_0^2)=(0,1)}R_0) \neq \emptyset$.

Since $\sigma_{B_i=2}(Q(\sigma_{(A_0^1,A_0^2)=(2,2)}R_0)) \neq \emptyset$ and $\sigma_{B_i=3}(Q(\sigma_{(A_0^1,A_0^2)=(3,3)}R_0)) \neq \emptyset$, we have that R_0 contains $\{(2,2),(3,3)\}$ and all arguments in the head of Q must be free variables.
Thus, $Q(\sigma_{(A_0^1,A_0^2)=(0,1)}R_0) \neq \emptyset$ implies R_0 contains $\{(0,1)\}$. Since the head of Q contains only free variables, $Q(\sigma_{(A_0^1,A_0^2)=(0,1)}R_0)$ always returns tuples over the domain $\{0,1\}$ when Φ returns a non-empty set. As for the righthandside term, we simply let

$$\Phi' \;=\; \lambda Q.\, \lambda R_1 ... \lambda R_p.\;\; \pi_X\big(\sigma_{c(\bar{v}_1,...,\bar{v}_p)}(R_1 \bowtie ... \bowtie R_p)\big).$$

The output of Φ' on some relations $R_1, ..., R_p$ is the set of assignments for the occurrences of the variables $\bar{x}$ obtained by projecting the consistent assignments from $R_1 \bowtie ... \bowtie R_p$. Intuitively, an assignment of Φ' corresponds to an assignment of variables in $\overline{\alpha}$.
We now prove that

$$\Phi \subseteq \Phi' \qquad \text{iff} \qquad \forall\, \theta_{\bar{x}}\; \exists\, \theta_{\bar{y}}\;\; \big(\theta_{\bar{x}}\theta_{\bar{y}} \vDash \bar{\alpha}\big).$$

$\Rightarrow$) Assume that $\Phi \subseteq \Phi'$ and let T be the set of all assignments θ for $\bar{x}$ and $\bar{y}$ that satisfy $\bar{\alpha}$. For every $1 \leq i \leq p$, we let R_i be the relation that consists of all tuples t for which there is $\theta \in T$ such that $t.v_i^j$ coincides with the value of the variable in α_i^j under the assignment θ. Clearly, $\pi_\emptyset\big(\sigma_{\bar{v}_i=t_{i,j}}(R_i)\big)$ evaluates to `true`, for all $1 \leq i \leq p$ and all $1 \leq j \leq 7$, and hence $\Phi(R_1, ..., R_p)$ evaluates to the set X of all possible consistent assignments for the occurrences of the variables $\{x_1, ..., x_m\}$ in the clauses $\alpha_1, ..., \alpha_p$. Since $\Phi(R_0, ..., R_p) \subseteq \Phi'(R_1, ..., R_p)$, we know that $\Phi'(R_1, ..., R_p)$ contains at least all tuples from $Q(\sigma_{(A_0^1,A_0^2)=(0,1)}R_0)$. This holds for every assignment of Q and R_0, so every assignment of $\theta_{\bar{x}}$ for $\bar{x}$ can be extended to an assignment θ for $\bar{x}$ and $\bar{y}$ satisfying $\bar{\alpha}$.

$\Leftarrow$) Suppose that every assignment $\theta_{\bar{x}}$ for $\bar{x}$ can be extended to an assignment θ for $\bar{x}$ and $\bar{y}$ satisfying $\bar{\alpha}$. Fix some generic relations $R_0, ..., R_p$ and assume, without loss of generality, that $\Phi(R_1, ..., R_p)$ evaluates to a *non-empty* set X (if this were not the case, then the inclusion $\Phi(R_1, ..., R_p) \subseteq \Phi'(R_1, ..., R_p)$ would follow trivially). Let t be a generic tuple in X. By construction, t represents an assignment $\theta_{\bar{x}}^t$ for the occurrences of the variables $\bar{x}$ in the clauses $\alpha_1, ..., \alpha_p$, which is obtained by projecting some generic (consistent) assignment from $Q(\sigma_{(A_0^1,A_0^2)=(0,1)}R_0)$ which is non-empty and ranges over $\{0,1\}$. From the hypothesis, we know that such an assignment $\theta_{\bar{x}}^t$ can be extended to an assignment θ^t for the occurrences of the variables $\bar{x}$ and $\bar{y}$ in such a way that all

clauses $\alpha_1, ..., \alpha_p$ are satisfied simultaneously. Finally, since $X \neq \emptyset$, we have that such an extension θ^t belongs to the product $R_1 \bowtie ... \bowtie R_p$ and hence $X \subseteq \Phi'(R_1, ..., R_p)$.

This completes the proof of the proposition. □

Of course, the hardness result does not hold in the symmetric case, where the lefthandside higher-order query has no occurrences of query variables:

Proposition 6.24. *The problem of deciding the containment* $\Phi \subseteq_{\lambda\mathsf{CQ}} \Phi'$*, where* $\Phi, \Phi' \in \mathsf{HO}_2^{\downarrow}[\mathsf{CQ}]$ *and* Φ *contains no occurrences of query variables, is NP-complete.*

Proof. When there is no query variable in either higher-order term, the containment problem above becomes the containment problem between two CQs, which is known to be NP-hard. By monotonicity, it suffices to show that containment holds when all the query variables in Φ' return $\emptyset$. If Φ' contains a query variable, it will also return $\emptyset$ in this case, and hence we need only check satisfiability of Φ, which is straightforward (and a special case of CQ containment). Otherwise, Φ' is just a CQ. In either case, we can reduce this problem to the containment problem between two CQs, which is in NP. □

As for the upper bounds, at the moment, we are only able to provide a result that matches with Proposition 6.23:

Proposition 6.25. *The problem of deciding the containment* $\Phi \subseteq_{\lambda\mathsf{CQ}} \Phi'$*, where* $\Phi, \Phi' \in \mathsf{HO}_2^{\downarrow}[\mathsf{CQ}]$ *and* Φ' *contains no occurrences of query variables, is in* Π_2^{P}*.*

Proof. Intuitively, the query variables over CQ signature are not able to return relational constants like those over RA^+ or RA. Thus, even when Φ' does not contain query variables, it is possible that $\Phi \subseteq_{\lambda\mathsf{CQ}} \Phi'$ as in the following example:

Example 17. *Given a relational variable* R *of type* (a, b) *and a query variable* Q *of type* $(a, b) \to (a, b)$*, consider two terms below:*

$$\begin{aligned} \Phi &= \lambda Q.\lambda R.\pi_{\emptyset} Q(\sigma_{a=b} R) \\ \Phi' &= \lambda Q.\lambda R.\pi_{\emptyset} R \end{aligned}$$

It is easy to see that $\Phi \subseteq_{\lambda\mathsf{CQ}} \Phi'$*.*

The proof of the above result is based on the idea that, in order to decide the containment $\Phi \subseteq_{\lambda\mathsf{CQ}} \Phi'$, it is sufficient to consider instantiations of query variables having size bounded by a polynomial in the size of the input terms.

Because the query variables only occur in Φ, the requirement on Φ becomes strictly stronger as we make each input query less restrictive. Consider an arbitrary query variable Q_i that occurs in Φ, assume that Q_i maps a set of relational types $\{\tau_1, \ldots, \tau_n\}$ into a relational type τ. Assume that Q_i has input variables $R_1, \ldots, R_n$ of types $\tau_1, \ldots, \tau_n$. The least restrictive queries one can consider are cross-products of relation symbols and constants. Therefore, to solve the containment problem we only need to consider all Q_i of this form. The size of those queries is bounded by a polynomial in the input size. Hence the number of new constants and variables in each Q_i is also bounded by polynomials. We can thus determine an assignment of Q_i by making a polynomial number of choices.
Below, we prove:

Let Φ_0 be the value of Φ when each query variable Q_i is instantiated with a given CQ. Deciding whether $\Phi_0 \subseteq \Phi'$ is in NP.

Note that Φ' is an ordinary conjunctive query, while Φ_0 is a composition of conjunctive queries. We can look for a homomorphism of Φ' in Φ_0 without fully expanding the composition, using a polynomial number of choices. □

We now consider another case of higher-order queries over the CQ signature, where query variables occur in both sides of the containment and the containment problem is in Π_2^{P}. In general, with the CQ signature, we cannot use the same method as before to show that the higher-order containment problem is in Π_2^{P}. Here we show the upper bound for the case where all query variables are boolean. A simple example of this case is the containment of $\lambda Q.(Q(\tau_1) \wedge Q(\tau_2))$ in $\lambda Q.Q(\rho_1)$ with τ_1, τ_2 and ρ_1 order 0 terms. Let us recall from Definition 8 that a term is un-nested if a variable never occurs as an argument of itself in the term.

Proposition 6.26. *The problem of deciding containment between two un-nested $\mathsf{HO}_2^{\downarrow}[\mathsf{CQ}]$ terms with only boolean query variables is NP-complete.*

Proof. The lower bound is obtained via reduction from the containment problem between two conjunctive queries.

For simplicity, we show the upper bound for the case where there exists only one query variable Q. The general case can be extended easily from this. Since there is no nesting of query variables and Q is boolean, the possible operators outside an application of Q are $\times$ and $\wedge$. When one term is boolean, the $\wedge$ operator requires

that the other term is also boolean, and has the same semantics as $\times$. Thus, we only use the $\wedge$ operator over two boolean subterms. In addition, we can assume without loss of generality that Q has only one relation as input. Two normalized order 2 terms Φ and Φ' are thus of the following form

$$\begin{aligned}\Phi &= \lambda Q.\lambda \vec{R}.\tau_0 \times Q(\tau_1) \wedge Q(\tau_2) \wedge \ldots \wedge Q(\tau_n)\\ \Phi' &= \lambda Q.\lambda \vec{R}.\rho_0 \times Q(\rho_1) \wedge Q(\rho_2) \wedge \ldots \wedge Q(\rho_m)\end{aligned}$$

with $\vec{R}$ the set of all relational variables occurring in τ_i and ρ_i.

We show that if the containment holds for all boolean Q of small size, it will actually hold for all boolean Q.

Each τ_i, ρ_i can be transformed into relational calculus, yielding formulas of the form:

$$\begin{aligned}\tau_i &= \{(\vec{x}_i) \mid \exists \vec{y}_i.\vec{A}_i(\vec{x}_i, \vec{y}_i)\}\\ \rho_i &= \{(\vec{s}_i) \mid \exists \vec{t}_i.\vec{B}_i(\vec{s}_i, \vec{t}_i)\}\end{aligned}$$

with $\vec{A}_i(\vec{x}_i, \vec{y}_i)$ and $\vec{B}_i(\vec{s}_i, \vec{t}_i)$ conjunctions of atoms with arguments being constants and variables respectively in $(\vec{x}_i, \vec{y}_i)$ and in $(\vec{s}_i, \vec{t}_i)$.

We consider all assignments Q_0 for Q of the following form.

$$Q_0 = \lambda R.\{() \mid \exists \vec{u}_0.R(\vec{u}_0)\}$$

Then $\Phi(Q_0)$ and $\Phi'(Q_0)$ are of the form below.

$$\begin{aligned}\Phi(Q_0) &= \{\vec{x}_0 \mid \exists \vec{u}_0 \vec{y}_0 \ldots \vec{y}_n.\vec{A}_0(\vec{x}_0, \vec{y}_0), \vec{A}_1(\vec{x}_1, \vec{y}_1)[\vec{x}_1/\vec{u}_0], \ldots, \vec{A}_n(\vec{x}_n \vec{y}_n)[\vec{x}_n/\vec{u}_0]\}\\ \Phi'(Q_0) &= \{\vec{s}_0 \mid \exists \vec{u}_0 \vec{t}_0 \ldots \vec{t}_m.\vec{B}_0(\vec{s}_0, \vec{t}_0), \vec{B}_1(\vec{s}_1, \vec{t}_1)[\vec{s}_1/\vec{u}_0], \ldots, \vec{B}_m(\vec{s}_m \vec{t}_m)[\vec{s}_m/\vec{u}_0]\}\end{aligned}$$

There exists a homomorphism h_0 from $\Phi'(Q_0)$ to $\Phi(Q_0)$ such that $h_0(\vec{s}_0) = \vec{x}_0$ and $h_0(\vec{u}_0) = \vec{u}_0$. The $h_0(\vec{u}_0) = \vec{u}_0$ condition holds because by choosing Q_0, we can assign each u_0^i in $\vec{u}_0$ a constant c_i that does not occur in $\Phi(Q_0)$ and $\Phi'(Q_0)$, then $h_0(u_0^i) = u_0^i$ for all i.

An arbitrary assignment Q_k to Q is of the form $\lambda R.\{() \mid \exists \vec{u}_1 \ldots \vec{u}_k.R(\vec{u}_1), \ldots, R(\vec{u}_k)\}$,

Φ and Φ' are of the following form:

$$\begin{aligned}
\Phi(Q_k) &= \{\vec{x}_0 \mid \exists \vec{y}_0.\vec{A}_0(\vec{x}_0,\vec{y}_0),\\
&\quad \exists \vec{u}_1\vec{y}_1^1 \ldots \vec{y}_n^1.\vec{A}_1(\vec{x}_1^1,\vec{y}_1^1)[\vec{x}_1^1/\vec{u}_1],\ldots,\vec{A}_n(\vec{x}_n^1,\vec{y}_n^1)[\vec{x}_n^1/\vec{u}_1]\\
&\quad \ldots\\
&\quad \exists \vec{u}_k\vec{y}_1^k \ldots \vec{y}_n^k.\vec{A}_1(\vec{x}_1^k,\vec{y}_1^k)[\vec{x}_1^k/\vec{u}_k],\ldots,\vec{A}_n(\vec{x}_n^k,\vec{y}_n^k)[\vec{x}_n^k/\vec{u}_k]\}\\
\Phi'(Q_k) &= \{\vec{s}_0 \mid \exists \vec{t}_0.\vec{B}_0(\vec{s}_0,\vec{t}_0),\\
&\quad \exists \vec{u}_1\vec{t}_1^1 \ldots \vec{t}_m^1.\vec{B}_1(\vec{s}_1^1,\vec{t}_1^1)[\vec{s}_1^1/\vec{u}_1],\ldots,\vec{B_m}(\vec{s}_m^1,\vec{t}_m^1)[\vec{s}_m^1/\vec{u}_1]\\
&\quad \ldots\\
&\quad \exists \vec{u}_k\vec{t}_1^k \ldots \vec{t}_m^k.\vec{B}_1(\vec{s}_1^k,\vec{t}_1^k)[\vec{s}_1^k/\vec{u}_k],\ldots,\vec{B}_n(\vec{s}_m^k,\vec{t}_m^k)[\vec{s}_m^k/\vec{u}_k]\}
\end{aligned}$$

In the formula above, we can choose $\vec{t}_i^j$ such that $\{\vec{t}_1^i,\ldots,t_m^i\} \cap \{\vec{t}_1^j,\ldots,t_m^j\} = \emptyset$ for all $i \neq j$.

From the result above we know there exist mappings $h_1,\ldots,h_k$ such that:

- $h_i(\vec{s}_0) = \vec{x}_0$ for all $i \in (1,k)$,
- $h_i(\vec{u}_i) = \vec{u}_i$ for all $i \in (1,k)$,
- $h_i(\vec{t}_j^i) \in \{y_1^j,\ldots,y_n^j\}$ for all $i \in (1,k)$, $j \in (1,m)$.

We build $h = \bigcup_{i\in\{1,\ldots,k\}} h_i$ and show that h is a homomorphism from Φ' to Φ. We now need to make sure that if $x = y$, then $h(x) = h(y)$. For all x, y that are in $\{\vec{s}_0,\vec{u}_1,\ldots,\vec{u}_k\}$, we have $h(x) = x$ and $h(y) = y$; thus $h(x) = h(y)$.

For all the equivalent pairs of x, y that are not in $\{\vec{s}_0,\vec{u}_1,\ldots,\vec{u}_k\}$, x, y must be in $\{\vec{t}_1^i,\ldots,\vec{t}_m^i\}$ for some $i \in (1,k)$ because we have chosen $\vec{t}_i^j$ such that for all $i \neq j$, $\{\vec{t}_1^i,\ldots,t_m^i\} \cap \{\vec{t}_1^j,\ldots,t_m^j\} = \emptyset$. Thus, $h(x) = h_i(x)$ and $h(y) = h_i(y)$. From the property of mapping h_i, when $x = y$, we have $h_i(x) = h_i(y)$, i.e. $h(x) = h(y)$.

Since the number of assignments of the form Q_0 for Q is polynomial in the input size, we can polynomially reduce the containment problem to the regular containment problem between two conjunctive queries. This means that the problem is in NP. □

To demonstrate the proof above, we consider the following example.

Example 18. *Let* $\Phi = Q(\tau_1) \times Q(\tau_2)$, $\Phi' = Q(\rho_1)$ *with:*

$$\begin{aligned}
\tau_1 &= \{(x_1,y_1) \mid R_1(x_1,y_1)\}\\
\tau_2 &= \{(x_2,y_2) \mid R_2(x_2,y_2)\}\\
\rho_1 &= \{(s_1,t_1) \mid \exists s,t.R_1(s_1,s),R_2(t_1,t)\}
\end{aligned}$$

By choosing $Q_0 = \lambda R.\{() \mid R(a,b)\}$ *with* a, b *two constants, we reduce the given containment problem to checking the containment* $\Phi(Q_0) \subseteq \Phi'(Q_0)$*, which is equivalent to:*

$$\{() \mid R_1(a,b), R_2(a,b)\} \subseteq \{() \mid \exists s,t.R_1(a,s), R_2(b,t)\}$$

Since there does not exist a homomorphism from $\Phi'(Q_0)$ *into* $\Phi(Q_0)$*, we can conclude that* Φ *is not contained in* Φ'*.*

6.4 Containment between HOCVP terms of order 1

This section considers the containment problem between higher-order queries over complex values. We start with order 1 terms, which are polynomially reducible to Monad Algebra expressions [Koc06] (see also in Subsection 4.2.3). Thus, the containment problem is equivalent to the containment problem between Monad Algebra expressions, which has not been explicitly studied.

Proposition 6.27. *The containment problem* $\Phi \subseteq_{\lambda\mathsf{MA}^+} \Phi'$*, where* Φ, Φ' *are* HOCVP^0 *terms of order 1, is* NEXPTIME*-hard and in* $TA(2^{O(n)}, 1)$*.*

Proof. Due to the polynomial reduction between HOCVP^0 and Monad Algebra, the lower bound is obtained from the NEXPTIME-hardness of evaluating Monad Algebra with atomic equivalence and without negation.

To show the upper bound, we consider the non-containment problem between Φ and Φ'. This is obtained by adapting the NEXPTIME-membership proof in Koch's paper [Koc06].

A nested relation R is equivalent to a tree, which is represented by a set of paths from the root to the leaves. We inductively define a function *path* that transforms a nested relation to a set of paths as follows.

- If n is an atomic value, then $path(n) = \{1.n\}$.
- $path(\langle A_1 : t_1, \ldots, A_n : t_n\rangle) = \bigcup_{i\in(1,n)}\{A_i.l \mid l \in path(t_i)\}$,
- $path(\{t_1, \ldots, t_n\}) = \bigcup_{i\in(1,n)}\{i.l \mid l \in path(t_i)\}$.

We also inductively define a path query language HOCVP^0_{path} to query relations created by the *path* function above.

$$
\begin{aligned}
[\![c]\!](P) &:= \{m.c \mid m.p \in P\} \\
[\![\pi_A]\!](P) &:= \{m.p \mid m.A.p \in P\} \\
[\![sng]\!](P) &:= \{m.1.p \mid m.p \in P\} \\
[\![flatten]\!](P) &:= \{m.(i.j).p \mid m.i.j.p \in P\} \\
[\![A =_{atomic} B]\!](P) &:= \{m.1.\langle\rangle \mid m.A.p, m.B.p \in P\} \cup \{m.1 \mid m.A.p \in P\} \\
[\![\pi_A \cup \pi_B]\!](P) &:= \{m.(1.i).p \mid m.A.i.p \in P\} \cup \{m.(2.i).p \mid m.B.i.p \in P\} \\
[\![pairwith_{A_j}]\!](P) &:= \{m.i.A_j.p \mid m.A_j.i.p \in P\} \cup \\
&\quad \{m.i.A_k.p \mid m.A_j.i.p, m.A_k.p \in P \wedge j \neq k\} \\
[\![map(f)]\!](P) &:= [\![map_e]\!]([\![f]\!]([\![map_b]\!](P))) \\
[\![\langle A_1 : f_1, \ldots, A_k : f_k \rangle]\!](P) &:= \{m.A_1.p \mid m.p \in [\![f_1]\!](P)\} \cup \ldots \cup \\
&\quad \{m.A_k.p \mid m.p \in [\![f_k]\!](P)\} \\
[\![f(g)]\!](P) &:= [\![f]\!]([\![g]\!](P)) \\
[\![\lambda x.f]\!](P) &:= \lambda x.([\![f]\!](P))
\end{aligned}
$$

where

$$
\begin{aligned}
[\![map_b]\!](P) &:= \{(m.i).p \mid m.i.p \in P\} \\
[\![map_e]\!](P) &:= \{m.i.p \mid (m.i).p \in P\}
\end{aligned}
$$

and $(x.y)$ denotes a label that is uniquely generated and identified by x and y.

The details of the equivalence between evaluating HOCVP^0 on nested relations and HOCVP^0_{path} on the path representation of nested relations are described in [Koc06]. The main difference is that we use $\lambda x.f$ instead of the id operator.

We represent a HOCVP^0_{path} query by a construction tree ξ, which is similar to the construction trees for HOCV. Since there are only variables of order 0, the tree ζ obtained from ξ after β-reduction has size exponential in the size of ξ but has depth polynomial in the depth of ξ. We use notations $[\![\zeta]\!]$ and $[\![\zeta']\!]$ to respectively denote the *path* representations of the terms Φ and Φ' after β-reduction.

We show that if one expression is not contained in another, there is a small counterexample. We first show the following lemma:

Lemma 8. *Assuming there is an assignment D_0 to the relational variables such that $t_0 \in [\![\zeta]\!](D_0) - [\![\zeta']\!](D_0)$, we show that there exists D_1 that consists of only paths of length polynomial in the input size and $t_1 \in [\![\zeta]\!](D_1) - [\![\zeta']\!](D_1)$.*

Assuming h is the maximal height of ζ and ζ', we define a function f as follows.

$$f(S) = \{p_s \mid p_s.p_t \in S, |p_s.p_t| > N\} \cup \{p_s \mid p_s \in S, |p_s| \leq N\}$$

with S a set of paths, $|p_s|$ the length of p_s, and $N = 3 \cdot h$.

We now prove that if $E_1 = [\![\zeta]\!](D_1)$, then $f(E_1) = [\![\zeta]\!](f(D_1))$ by induction on the height of ζ. The base case is obvious, so we only show the induction case. Assuming P is a term of height k such that $f(P(D_P)) = P(f(D_p))$, we prove that $f(P'(D_P)) = P'(f(D_P))$ for the case that P' has one more operator above P. We only show this for a few operators in HOCVP^0_{path}; the other operators are similar.

- Case $P' = [\![c]\!](P)$: We have $E'_P = [\![c]\!](P(D_P)) = [\![c]\!](E_P) = \{m.c \mid m.p \in E_P\}$. We also have $P'(f(D_P)) = [\![c]\!](P(f(D_P))) = [\![c]\!](f(E_P)) = \{m.c \mid m.p \in f(E_P)\}$. From the definition of f, we have $f(E'_P) = \{p_s \mid p_s.p_t \in \{m.c \mid m.p \in E_P\}, |p_s| > N\} \cup \{p_s \mid p_s \in \{m.c \mid m.p \in E_P\}, |p_s| \leq N\}$, which is equal to $\{m.c \mid m.p \in E_P\}$ because $|m.c| < N$. Additionally, $P'(f(D_P)) = \{m.c \mid m.p \in \{p_s \mid p_s.p_t \in E_P, |p_s| > N\} \cup \{p_s \mid p_s \in E_P, |p_s| \leq N\}\}$, which is also equal to $\{m.c \mid m.p \in E_P\}$.

- Case $P' = [\![flatten]\!](P)$: We have $E'_P = [\![flatten]\!](P(D_P)) = [\![flatten]\!](E_P) = \{m.(i.j).p \mid m.i.j.p \in E_P\}$. We also have $P'(f(D_P)) = [\![flatten]\!](P(f(D_P))) = [\![flatten]\!](f(E_P)) = \{m.(i.j).p \mid m.i.j.p \in f(E_P)\}$. From the definition of f, we easily obtain $f(E'_P) = P'(f(D_P))$.

From the result above, we have $f(\{t_0\}) \in [\![\zeta]\!](f(D_0)) - [\![\zeta']\!](f(D_0))$. Thus, by choosing t_1 be the element in $f(\{t_0\})$, and $D_1 = h(D_0)$, we obtain t_1 and D_1 that satisfy the lemma.

From Lemma 8, we can assume that there is an assignment D_1 to the relational variables such that $t_1 \in [\![\zeta]\!](D_1) - [\![\zeta']\!](D_1)$ and D_1, t_1 consists of paths of length polynomial in the input size. We show that there exists $D_2 \subseteq D_1$ such that D_2 consists of an exponential number of paths and $t_1 \in [\![\zeta]\!](D_2)$ by induction on the height of ζ. The base case is readily shown because each operator in the *path* query language requires at most two paths as its input to return one path as the output. The induction case is shown by again using the fact that each operator in the *path* query language requires at most two paths as its input to return one path as the output. Thus, ζ returns t_1 when the operators at the leaves have an exponential number of paths as their inputs.

In addition, $t_1 \notin [\![\zeta']\!](D_2)$ because of $D_2 \subseteq D_1$. Therefore, to show that $[\![\zeta]\!] \not\subseteq_{\lambda\mathsf{MA}^+} [\![\zeta']\!]$, we only need to consider witnesses D_2 consisting of an exponential number of paths of polynomial size.

Since the query complexity of evaluating Φ and Φ' is in NEXPTIME and t_1 is of polynomial size, the query complexity of checking $t_1 \in [\![\zeta]\!](D_2) - [\![\zeta']\!](D_2)$ is in $\mathrm{TA}(2^{O(n)}, 1)$. In addition, we know that the data complexity of evaluating Monad Algebra is in PSPACE [Koc06], and the size of D_2 is exponential in the input size. Therefore, we can check the non-containment problem using an alternating TM with one alternation from an existential state to a universal state and exponential running time. Thus, the containment problem is also in $\mathrm{TA}(2^{O(n)}, 1)$, which allows one alternation. □

Proposition 6.28. *The containment problem* $\Phi \subseteq_{\lambda\mathsf{MA}^+} \Phi'$*, where* $\Phi, \Phi' \in \mathsf{HOCVP}^1$*, is co-2-*NEXPTIME*-complete.*

Proof. The co-2-NEXPTIME-hardness of the containment problem between HO terms containing order 1 variables in Theorem 6.5 implies the lower bound in the proposition.

The upper bound is proved by showing that the non-containment problem is in 2-NEXPTIME for the language over *path* representations of nested relations. By β-reducing query variables in both queries, we obtain two queries Φ_0 and Φ_0' of exponential size in the input size. Assume that the construction trees of Φ_0 and Φ_0' are respectively ζ and ζ'; the *path* representations of Φ_0 and Φ_0' are respectively $[\![\zeta]\!]$ and $[\![\zeta']\!]$. From the proof of Proposition 6.27, we know if two expressions are not contained in one another, there is a small counterexample D_0, which is a set of at most doubly exponential number of paths but the length of paths is at most exponential in the input size. There also exists a path of length exponential in the input size such that $t \in [\![\zeta]\!](D_0) - [\![\zeta']\!](D_0)$.

By Theorem 5.28, the query complexity of evaluating Φ and Φ' is in EXPSPACE. Together with the fact that t is of exponential size, we have the query complexity of checking $t \in [\![\zeta]\!](D_0) - [\![\zeta']\!](D_0)$ is in EXPSPACE. In addition, we know that the data complexity of evaluating $[\![\zeta]\!](D_0)$ and $[\![\zeta']\!](D_0)$ is polynomial in the size of D_0, which is doubly exponential in the input size.

Thus the non-containment problem is in 2-NEXPTIME, i.e. the containment problem is in co-2-NEXPTIME. □

From the proposition above, by applying β-reduction and using the lower bound in Theorem 6.6, we have a general result below.

Theorem 6.29. *The containment problem* $\Phi \subseteq_{\lambda\mathsf{MA}^+} \Phi'$*, where* $\Phi, \Phi' \in \mathsf{HOCVP}^{(k-1)}$*, is co-*$k$-NEXPTIME*-complete.*

Containment between normalized HOCVP terms of order 1.
In the previous section, we did not consider the containment problem for normalized HO terms of order 1 because normalized HO terms of order 1 are just Relational Algebra expressions, for which the complexity of containment is known. Here we consider this problem for normalized HOCVP terms of order 1.

Theorem 6.30. *The containment problem* $\Phi \subseteq_{\lambda\mathsf{MA}^+} \Phi'$*, where* Φ *and* Φ' *are normalized boolean* HOCVP *terms of order 1, is* Π_2^{P}*-complete.*

Proof. The lower bound is obtained from the complexity of containment between positive Relational Algebra expressions. The upper bound is shown using the *path* function introduced above. Intuitively, normalized terms do not allow reuse of subterms in operators, so one can not use normalized terms to build a nested relation of doubly exponential size.

We need some basic facts about the transformation of a given HOCVP_{path} query into an equivalent union of HOCVP_{path} queries without Union. Such a transformation, which may lead to an exponential blow-up, is achieved by "pushing upward" all occurrences of the union operator of the Relational Algebra. Formally, the transformation rules are as follows:

$$
\begin{aligned}
[\![c]\!](P_1 [\![\cup]\!] P_2) &\rightsquigarrow [\![c]\!](P_1) \quad [\![\cup]\!] \quad [\![c]\!](P_2) \\
[\![\pi_A]\!](P_1 [\![\cup]\!] P_2) &\rightsquigarrow [\![\pi_A]\!](P_1) \quad [\![\cup]\!] \quad [\![\pi_A]\!](P_2) \\
[\![sng]\!](P_1 [\![\cup]\!] P_2) &\rightsquigarrow [\![sng]\!](P_1) \quad [\![\cup]\!] \quad [\![sng]\!](P_2) \\
[\![flatten]\!](P_1 [\![\cup]\!] P_2) &\rightsquigarrow [\![flatten]\!](P_1) \quad [\![\cup]\!] \quad [\![flatten]\!](P_2) \\
[\![A =_{atomic} B]\!](P_1 [\![\cup]\!] P_2) &\rightsquigarrow [\![A =_{atomic} B]\!](P_1) \quad [\![\cup]\!] \quad [\![A =_{atomic} B]\!](P_2) \\
[\![pairwith_{A_j}]\!](P_1 [\![\cup]\!] P_2) &\rightsquigarrow [\![pairwith_{A_j}]\!](P_1) \quad [\![\cup]\!] \quad [\![pairwith_{A_j}]\!](P_2) \\
[\![map(f)]\!](P_1 [\![\cup]\!] P_2) &\rightsquigarrow [\![map(f)]\!](P_1) \quad [\![\cup]\!] \quad [\![map(f)]\!](P_2) \\
[\![\langle A_1 : f_1, \ldots, A_k : f_k \rangle]\!](P_1 [\![\cup]\!] P_2) &\rightsquigarrow [\![\langle A_1 : f_1, \ldots, A_k : f_k \rangle]\!](P_1) \quad [\![\cup]\!] \\
&\qquad\qquad [\![\langle A_1 : f_1, \ldots, A_k : f_k \rangle]\!](P_2)
\end{aligned}
$$

By repeatedly applying these rules, one can transform any HOCVP_{path} query Φ into an equivalent union of HOCVP_{path} queries without Union of the form $\Phi = \Phi_1 \cup \ldots \cup \Phi_N$. This transformation, called *flattening*, has N bounded by an exponential in the size of Φ, and each Φ_i bounded by the size of Φ. Similarly, we have $\Phi' = \Phi'_1 \cup \ldots \cup \Phi'_M$.

Now we show that the non-containment problem between Φ and Φ' is in Σ_2^{P}. Assume that there exists a set of *path* relations $\bar{\mathcal{R}}$ that witnesses the non-containment:

$$\Phi_1(\bar{\mathcal{R}}) \cup ... \cup \Phi_N(\bar{\mathcal{R}}) \nsubseteq \Phi'_1(\bar{\mathcal{R}}) \cup ... \cup \Phi'_M(\bar{\mathcal{R}})$$

This means there exists i_0 and a path p such that

$$p \in \Phi_{i_0}(\bar{\mathcal{R}}) - \Phi'_1(\bar{\mathcal{R}}) \cup ... \cup \Phi'_M(\bar{\mathcal{R}})$$

Since the length of Φ_{i_0} is polynomial in the input size, by Lemma 8, a witness $\bar{\mathcal{R}}_0$ for $p \in \Phi_{i_0}(\bar{\mathcal{R}}) - \Phi'_1(\bar{\mathcal{R}}) \cup ... \cup \Phi'_M(\bar{\mathcal{R}})$ and p consist of paths of polynomial size in the input size. Each operator in Φ_{i_0} requires that the input contains at most one more path than the output. Together with the polynomial size of $\Phi_{i_0}(\bar{\mathcal{R}})$, thus, we only need to check for witnesses $\bar{\mathcal{R}}_1 \subseteq \bar{\mathcal{R}}_0$ of polynomial size, which leads to verifying $p \in \Phi_{i_0}(\bar{\mathcal{R}}_1)$ can be done in NP.

Checking $t \notin \Phi'_1(\bar{\mathcal{R}}_0) \cup ... \cup \Phi'_M(\bar{\mathcal{R}}_0)$ can be done by verifying $t \notin \Phi'_i(\bar{\mathcal{R}}_0)$ for all $i \in (1, M)$. We check $t \notin \Phi'_i(\bar{\mathcal{R}}_0)$ by considering all mappings of *path* queries in Φ'_i into paths in $\bar{\mathcal{R}}_1$. Thus the non-containment is in Σ_2^{P}, i.e. the containment is in Π_2^{P}. □

6.5 Containment between normalized HOCVP terms of order 2

This section studies the complexity of the containment problem for order 2 HOCVP terms in normal form. Namely, for higher-order queries, where the formal arguments are query variables and nested relational variables.

Theorem 6.31. *The containment problem $\Phi \subseteq_{\lambda\mathsf{MA}^+} \Phi'$, where Φ and Φ' are normalized* HOCVP *terms of order 2, is Π_2^{P}-complete.*

Proof. The lower bound is obtained from the Theorem 6.8. To show the upper bound, we use the *path* function and we adapt the proof for Theorem 6.8.

As in Proposition 6.10, we need the following "quantifier elimination" result for monotone queries, stating that the existence of a HOCVP_{path} query satisfying certain equalities between input and output sets of paths reduces to a boolean combination of containments between these sets.

Proposition 6.32. *For each i in $\{1, ..., m\}$, let $\mathtt{S}_i$ and $\mathtt{T}_i$ be a set of paths. The following properties are equivalent:*

1. *there exist a* HOCVP_{path} *query that is a union of* $\mathtt{Q}_1, \ldots, \mathtt{Q}_m$*, such that* $\mathtt{Q}_i(\mathtt{S}_i) = \mathtt{T}_i$ *for all* $i \in \{1, ..., m\}$*;*

2. *for every pair of indices* $i, j \in \{1, ..., m\}$*, if* $\mathtt{S}_i \subseteq \mathtt{S}_j$*, then* $\mathtt{T}_i \subseteq \mathtt{T}_j$*.*

Proof. The implication from 1. to 2. is trivial from the monotonicity of queries without negation. The implication from 2. to 1. is proved as follows. First, we introduce, for every index $i \in \{1, ..., k\}$, a union of queries that, given a set of paths $\mathtt{S}$, returns either $\mathtt{T}_i$ or the empty relation, depending on whether or not $\mathtt{S}$ contains all paths of $\mathtt{S}_i$. □

We now use the transformation *path* to prove the theorem. Let us fix two order 2 terms in normal form:

$$\Phi = \lambda Q_1 \ldots \lambda Q_m.\ \lambda R_1 \ldots \lambda R_n.\ \tau$$
$$\Phi' = \lambda Q_1 \ldots \lambda Q_m.\ \lambda R_1 \ldots \lambda R_n.\ \tau'$$

where each Q_i is an order 1 query variable, each R_j is an order 0 variable, and τ, τ' are well-typed HOCVP_{path} of order 0 over the variables $Q_1, ..., Q_m$, $R_1, ..., R_n$ and order 0 constants, which are sets of paths. We reduce the non-containment relationship $\Phi \not\subseteq_{\lambda\mathsf{RA}^+} \Phi'$ to the existence of some queries $\mathtt{Q}_1, ..., \mathtt{Q}_m$ over *path* operators and some $\mathtt{R}_1, ..., \mathtt{R}_n$ that witness $[\![\tau]\!](\bar{\mathtt{Q}}, \bar{\mathtt{R}}) \not\subseteq [\![\tau']\!](\bar{\mathtt{Q}}, \bar{\mathtt{R}})$.

As in the proof of Theorem 6.8 concerning containment between normalized HO terms of order 2, we introduce new nested relations for the intermediate outputs produced by the subterms of τ and τ'. We enumerate all occurrences of proper subterms of τ that are *arguments to a query variable* Q_i, for some $1 \leq i \leq m$. Let $\sigma_1, ..., \sigma_k$ be such an enumeration such that $j < j'$ holds whenever σ_j occurs inside $\sigma_{j'}$. We then associate with each occurrence σ_j the following objects: (i) the index $i_j \in \{1, ..., m\}$ of the query variable to which σ_j is applied, (ii) two nested relations $\mathtt{S}_j, \mathtt{T}_j$ (of appropriate types), (iii) a term $\mathtt{P}_j$ obtained from σ_j by replacing any top-level subterm of the form $Q_{i'}(\sigma_{j'})$ by $\mathtt{T}_{j'}$. We further introduce an additional query constant $\mathtt{P}_0$, obtained from τ by replacing any top-level subterm of the form $Q_{i'}(\sigma_{j'})$ by $\mathtt{T}_{j'}$. This means that each term $\mathtt{P}_j$, with $0 \leq j \leq k$, is a normalized HOCVP^0 term over the nested relations $\mathtt{R}_1, ..., \mathtt{R}_m$, $\mathtt{T}_1, ..., \mathtt{T}_k$. We also give similar definitions for the objects i'_j, $\mathtt{S}'_j$, $\mathtt{T}'_j$, $\mathtt{P}'_j$ with respect to the occurrences of subterms in τ'.

From the notations above, we reduce the non-containment problem for the HOCVP_{path} case to checking the following:

$$\begin{aligned}
&\exists\, Q_1, ..., Q_m \\
&\exists\, \bar{R}, \bar{S}, \bar{T}, \bar{S}', \bar{T}'. \quad \mathtt{P}_0(\bar{R}, \bar{T}, \bar{T}') \not\subseteq \mathtt{P}'_0(\bar{R}, \bar{T}, \bar{T}') \;\; \wedge \\
&\bigwedge_{1\leq j\leq k} \mathtt{P}_j(\bar{R}, \bar{T}, \bar{T}') = S_j \;\; \wedge \;\; \bigwedge_{1\leq j\leq h} \mathtt{P}'_j(\bar{R}, \bar{T}, \bar{T}') = S'_j \;\; \wedge \\
&\bigwedge_{1\leq j\leq k} Q_{i_j}(S_j) = T_j \qquad \wedge \;\; \bigwedge_{1\leq j\leq h} Q_{i'_j}(S'_j) = T'_j.
\end{aligned} \tag{6.6}$$

By exploiting Proposition 6.32, we can get rid of the existential quantification over $Q_1, ..., Q_m$ thus obtaining:

$$\begin{aligned}
&\exists\, \bar{R},\, \bar{S},\, \bar{T},\, \bar{S}',\, \bar{T}'. \quad \mathtt{P}_0(\bar{R}, \bar{T}, \bar{T}') \not\subseteq \mathtt{P}'_0(\bar{R}, \bar{T}, \bar{T}') \;\; \wedge \\
&\bigwedge_{1\leq j\leq k} \mathtt{P}_j(\bar{R}, \bar{T}, \bar{T}') = S_j \quad \wedge \;\; \bigwedge_{1\leq j\leq h} \mathtt{P}'_j(\bar{R}, \bar{T}, \bar{T}') = S'_j \;\; \wedge \\
&\bigwedge_{1\leq j,j'\leq k} S_j \subseteq S_{j'} \rightarrow T_j \subseteq T_{j'} \wedge \bigwedge_{1\leq j,j'\leq h} S'_j \subseteq S'_{j'} \rightarrow T'_j \subseteq T'_{j'} \;\; \wedge \\
&\bigwedge_{\substack{1\leq j\leq k \\ 1\leq j'\leq h}} S_j \subseteq S'_{j'} \rightarrow T_j \subseteq T'_{j'} \wedge \bigwedge_{\substack{1\leq j\leq h \\ 1\leq j'\leq k}} S'_j \subseteq S_{j'} \rightarrow T'_j \subseteq T_{j'}.
\end{aligned} \tag{6.7}$$

We rename the relational variables T_j and $T'_{j'}$, where j ranges over $\{1, ..., k\}$ and j' ranges over $\{1, ..., h\}$, by new relational variables U_i, where i ranges over an appropriate set I of indices isomorphic to $\{1, ..., k\} \uplus \{1, ..., h\}$, and, similarly, replace the queries $\mathtt{P}_j(\bar{R}, \bar{T}, \bar{T}')$ and $\mathtt{P}'_j(\bar{R}, \bar{T}, \bar{T}')$ by new queries $\mathtt{O}_i(\bar{R}, \bar{U})$. Accordingly, the conditions of the form $S_j \subseteq S_{j'} \rightarrow T_j \subseteq T_{j'}$ will be replaced by equivalent conditions of the form $\mathtt{O}_i(\bar{R}, \bar{U}) \subseteq \mathtt{O}_{i'}(\bar{R}, \bar{U}) \rightarrow U_i \subseteq U_{i'}$, where the pair (i, i') is either $(0, 0)$ or an element of an appropriate subset D of $I \times I$.
Now, for every partition $\mathcal{D} = (D_+, D_-)$ of D, we denote by $\Sigma_{\mathcal{D}}$ the set of all positive constraints of the form $U_i \subseteq U_{i'}$, with $(i, i') \in D_+$, and all negative constraints of the form $U_i \not\subseteq U_{i'}$, with $(i, i') \in D_-$. Intuitively, each $\Sigma_{\mathcal{D}}$ is a maximal set of containment relationships between the various instances U_i and $U_{i'}$, for all $(i, i') \in D$. Therefore, Property (6.7) holds iff there exist a partition $\mathcal{D} = (D_+, D_-)$ of D such that

$$\exists\, \bar{R}, \bar{U} \vDash \Sigma_{\mathcal{D}}. \;\; \mathtt{O}_0(\bar{R}, \bar{U}) \not\subseteq \mathtt{O}'_0(\bar{R}, \bar{U}) \;\; \wedge \bigwedge_{(i,i')\in D_-} \mathtt{O}_i(\bar{R}, \bar{U}) \not\subseteq \mathtt{O}_{i'}(\bar{R}, \bar{U}). \tag{6.8}$$

We observe that any containment relationship of the form $\mathsf{O}_i(\bar{R}, \bar{U}) \not\subseteq \mathsf{O}'_{i'}(\bar{R}, \bar{U})$, where O_i is a normalized HOCVP_{path} term of order 1, is equivalent to an existential quantification over all containment relationships of the form $\tilde{\mathsf{O}}_{i,l}(\bar{R}, \bar{U}) \not\subseteq \mathsf{O}'_{i'}(\bar{R}, \bar{U})$, where $\tilde{\mathsf{O}}_{i,l}$ is a conjunct of the flattening of O_i.

We show that the satisfiability problem of Equation 6.8 is in Σ_2^{P} by giving the witness for the conjunction of non-containments under containment and non-containment constraints. Let D_i be an instance for each U_i and $\mathcal{D}_0$ be a substitution for $\mathcal{D}$ that makes Equation 6.8 satisfiable. We only need witnesses D_i^* of polynomial size to witness the non-containment. To make these witnesses satisfy a non-containment constraint of the form $U_i \not\subseteq U_j$ in $\mathcal{D}_-$, we add one item from $\bigcup D_i - \bigcup D_i^*$ to D_i^*. Then to make a containment constraint of the form $U_i \subseteq U_j$ in $\mathcal{D}_+$ satisfiable, we use the new instance D_j^{**} with $D_j^{**} = D_j^* \cup D_i^*$ for R_j.

We also guess a mapping from $\tilde{\mathsf{O}}_i(\bar{R}, \bar{U})$ to the witness and then check the satisfiability of the witness in polynomial time. In order to verify if these small witnesses satisfy that $t_i \notin \mathsf{O}'_0(\bar{R}, \bar{U})$, we check all the possible mappings from queries to paths, which is in coNP.

Thus the satisfiability of Equation 6.8 can be solved in Σ_2^{P}, i.e. the higher-order containment problem is in Π_2^{P}. □

6.6 Conclusions

This chapter has considered two containment problems: query containment, where two terms are of order 1, and higher-order containment, where two terms are of order 2. The chapter considers the relational higher-order language HO in detail, then extends the results to the complex-valued higher-order language HOCVP.

The complexity results are given based on the highest order of variables in the terms. We have given complexity results for containment between unnormalized order 1 terms containing variables of arbitrary order for both HO and HOCVP. These results are shown using a combination of β-reduction and classical techniques for the containment problem on lower order languages.

For the case of order 2 HO, we have given tight bounds on the complexity of equivalence for normal-form terms when the base is positive Relational Algebra and handled a number of special cases for other bases. Using similar techniques, we have shown tight bounds for the case of normalized HOCVP.

The containment problem for the relational case has been investigated in depth for different signatures, e.g., RA^+, RA, and CQ. Nevertheless, the containment problem

for the cases of XML and complex values still has a lot of open problems. More details on the open problems are presented in Chapter 9.

Chapter 7

Typing Problems

7.1 Introduction

In the previous chapters we have assumed that types are explicitly given for all variables and constants. Under these conditions, one can check in a linear scan that a term is well-formed and derive its type, just by applying the well-formedness rules. In this chapter, we look at the most basic typing problem, the typeability problem mainly for HO and HOCV. This problem is generalized to the *general typeability problem*, where some of the variables and constants of the term can be given types. This chapter will present a polynomial reduction from the typeability problem for our higher-order query langauges to the typeability problem for ordinary query languages. We also consider the complexity of evaluating weakly-typed HO terms, where only the types of relational constants in the terms are given.

7.2 Typeability for higher-order languages

A higher-order term is well-typed when all its subterms satisfy typing rules for constants and for λ-calculus. Before defining the general typeability problem, we give the following set of typing rules for HOCV. The set of typing rules for HO is given in Subsection 4.1.1.

$$\frac{\Gamma \vdash x : \mathcal{T} \qquad \mathtt{order}(\mathcal{T}) = 0}{\Gamma \vdash sng(x) : \{\mathcal{T}\}}$$

$$\frac{\Gamma \vdash X : \{\{\mathcal{T}\}\} \qquad \mathtt{order}(\{\{\mathcal{T}\}\}) = 0}{\Gamma \vdash flatten(X) : \{\mathcal{T}\}}$$

$$\frac{\Gamma \vdash \tau : \langle A_1 : \mathcal{T}_1, \ldots, A_i : \{\mathcal{T}_i\}, \ldots, A_n : \mathcal{T}_n \rangle}{\Gamma \vdash pairwith_{A_i}(\tau) : \{\langle A_1 : \mathcal{T}_1, \ldots, A_i : \mathcal{T}_i, \ldots, A_n : \mathcal{T}_n \rangle\}}$$

$$\frac{\Gamma \vdash \tau : \langle A_1 : \mathcal{T}_1, \ldots, A_i : \mathcal{T}_i, \ldots, A_n : \mathcal{T}_n \rangle}{\Gamma \vdash \pi_{A_i}(\tau) : \mathcal{T}_i}$$

$$\frac{\Gamma \vdash \tau_1 : \mathcal{T} \qquad \Gamma \vdash \tau_2 : \mathcal{T} \quad \mathtt{order}(\mathcal{T}) = 0}{\Gamma \vdash \tau_1 \cup \tau_2 : \mathcal{T}}$$

$$\frac{\Gamma \vdash \tau : \langle A_1 : \mathcal{T}_1, \ldots, A_i : \mathcal{T}_i, \ldots, A_j : \mathcal{T}_j, \ldots, A_n : \mathcal{T}_n \rangle}{\Gamma \vdash \sigma_{A_i = A_j}(\tau) : \{\langle\rangle\}}$$

$$\frac{\forall i . \Gamma \vdash f_i : \mathcal{T}_i \qquad \forall i . \mathtt{order}(\mathcal{T}_i) = 0}{\Gamma \vdash \langle A_1 : f_1, \ldots, A_n : f_n \rangle : \langle A_1 : \mathcal{T}_1, \ldots, A_n : \mathcal{T}_n \rangle}$$

$$\frac{\Gamma \vdash X : \{\mathcal{T}\} \quad \mathtt{order}(\mathcal{T}) = 0 \quad \Gamma \vdash f : (\mathcal{T} \to \mathcal{T}') \quad \mathtt{order}(\mathcal{T}') = 0}{\Gamma \vdash map(f, X) : \{\mathcal{T}'\}}$$

$$\frac{\Gamma \vdash x : \mathcal{T} \qquad \Gamma \vdash \rho : \mathcal{T}'}{\Gamma \vdash \lambda x . \, \rho : (\mathcal{T} \to \mathcal{T}')}$$

$$\frac{\Gamma \vdash \tau : (\mathcal{T} \to \mathcal{T}') \qquad \Gamma \vdash \rho : \mathcal{T}}{\Gamma \vdash \tau(\rho) : \mathcal{T}}$$

We study the following typing problem:

Definition 14. General typeability problem:

Given a term and types for none or some of its variables and constants , check if there are type assignments to the remaining constants and variables such that all its subterms are well-typed.

We show that higher-order variables do not introduce additional complexity in typing higher-order complex-valued queries:

Theorem 7.1. *The general typeability problems for* HOCV *and for* HO *are polynomially reducible to the typing problems for Monad Algebra and for Relational Algebra, respectively.*

Proof. We give an algorithm that reduces the general typeability problem for a higher-order term τ_H to the typeability problem for a term without higher-order variables.

The algorithm begins by converting τ_H to a set of *equations between terms over type variables*, following the classical technique of Wand [Wan87b] for doing type inference in λ-calculi by producing a scheme of type equations. Terms are built up from basic types and type variables using the function constructor $\mathcal{T}$, with a type variable existing for every subterm of the query. When an application operator occurs that will bind a variable with a subterm, we add equations identifying the type of the variable with the type of the subterm. Formally, the set of equations is inductively built from the syntax of τ_H as follows. First we rename λ-variables to make sure that all λ-variables have different names. Then we apply the following steps.

- **Step 1:** Give a type name $\mathcal{T}_C$ for each constant or variable C.
- **Step 2:** Give a type name $\mathcal{T}_\tau$ for each subterm τ of τ_H.
- **Step 2:** Go through the subterms of τ_H to build a set of equations about the relationships between the types given in Step 1 and Step 2. There are two categories of subterms that give us equations. The first one is for subterms of τ_H that contain only contain constants, i.e., ordinary queries. We obtain the equations from the typing rules for the query operators. For example, a subterm $R_1 \cup R_2$ gives us the equation:

  $$\mathcal{T}_{R_1} = \mathcal{T}_{R_2}$$

  The second category is for subterms containing λ-application, which must satisfy the type matching conditions of β-reduction. If there is a subterm τ of the form $(\lambda X.\tau_1)(\tau_2)$, then we have three equations:

  $$\mathcal{T}_{\lambda X.\tau_1} = \mathcal{T}_X \to \mathcal{T}_{\tau_1}$$

  $$\mathcal{T}_\tau = \mathcal{T}_{\tau_1}$$

  $$\mathcal{T}_X = \mathcal{T}_{\tau_2}$$

We can solve the set of equations in linear time using a unification algorithm, e.g., the one of Paterson and Wegman [PW78]. The solution of the algorithm is a set of equalities on lower order types, which represent what is needed to ensure that the terms are typeable assuming all order 0 subterms are typeable (the notion of typeable makes sense equally for open terms). That is, we have constructed a set of equalities E between the types of order 0 subterms of τ_H, such that τ_H is typeable iff there is a type assignment to the order 0 subterms that makes each subterm typeable and additionally satisfies E. We abbreviate this by saying that $\tau_1 \ldots \tau_n$ are E-typeable.

We now claim that given order 0 terms $\tau_1 \ldots \tau_n$ and type equalities E, there is a single order 0 term τ such that τ is typeable iff $\tau_1 \ldots \tau_n$ are E-typeable. To make an equation $\mathcal{T}_{\tau_i} = \mathcal{T}_{\tau_j}$ in E satisfiable, we add a term $\tau_i \cup \tau_j$. Using this result, we can build an ordinary query that is typeable iff $\tau_1 \ldots \tau_n$ are E-typeable. This completes the reduction. □

From the above we get:

Proposition 7.2. *The general typeability problems for* HOCV *terms and for* HO *terms are* NP*-complete.*

Proof. The upper bound is obtained directly from the reduction in Theorem 7.1. The lower bound is proved by considering the order 0 variant of the problem. Vansummeren [Van05b] has shown that the typeability problem is NP-hard if we have projection, join, and selection with at least two base types. Since our higher-order query languages become ordinary query languages when there are not λ abstraction, the lower bound is shown. □

What about the XQuery variant XQH? Since we have abstracted away attributes in our core language, we cannot express constraints on lower-order types – thus, a similar reduction as above would give us the NP upper bound. Additionally, the argument of Van den Bussche et al. [VdBVGV05] would also give NP-hardness. On the other hand, a more realistic language would have attributes with multiple base data types in the data model, and supplement XQuery with an attribute axis and attribute equality selections in conditions.

7.3 Weakly-typed HO terms

We now introduce a weakly-typed language for HO, which allows us to build terms that have types with an 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 HO terms, defined in Subsection 4.1.1, 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 ρ is a term, then $\lambda X.\,\rho$ is a term; if τ and ρ 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 $T \to (T' \to T'')$ for some relational types T, T', T'', and consistent with function application and λ abstraction; e.g., if the type of ρ is T and the type of $\tau(\rho)$ is T', then the type of τ must be $T \to T'$.

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 $\mathsf{RA}_\times$, defined in Subsection 4.1.1, and we denote the resulting higher-order language by $\mathsf{HO}^-[\mathsf{RA}_\times]$.

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

Example 19. *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 0 term below.*

$$\tau_2 = \pi_{\{0,1\}} \left((\lambda R_2.(R_2 \mathsf{X} R_2)) \, (\lambda R_1.(R_1 \mathsf{X} R_1) D_0) \right)$$

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 $(\lambda R_1.(R_1 \mathsf{X} R_1) D_0)$, has four integer attributes.

We now consider the typing problem for weakly-typed degree 0 terms.

Proposition 7.3. *There exists a* PTIME *algorithm that takes a weakly-typed degree 0, 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 gives 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 τ_1 has type m and τ_2 type n, $\tau_1 \mathsf{X} \tau_2$ has type $m + n$.).
- For a subterm of the form $(\lambda R.\tau)\tau'$, if we have a type for τ' 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 7.4. *There is at most one consistent typing for a weakly-typed* HO *terms of degree* 0 *order* 0.

We now study evaluation for the weakly-typed HO terms. We will be concerned with the evaluation problem only for terms that have a consistent typing.

For a simply-typed HO term, we give types for all the variables and constants in the term. Thus, all the instances of the relational variables in the term are of exponential size in the term's size. In the case of weakly-typed HO, the product operator can build relational variables having an exponential number of attributes. This means that one can build relational instances of doubly-exponential size. This seems to increase the complexity of the evaluation problem; however, the next proposition shows that the complexity of evaluating degree 0 weakly-typed terms matches the simply-typed case.

Proposition 7.5. *Degree* 0 *weakly-typed* HO *terms can be evaluated in* PSPACE.

Proof. We give a recursive function Eval which takes as arguments a subterm of a given term τ and a set $S = \{(c_1, v_1) \dots (c_n, v_n)\}$, with c_i being positions in binary and v_i being values in the active domain of the term. The role of S is to keep track of only the attributes that are needed for evaluation, not all the attributes of the subterm, which can be exponential in the input size. By the evaluation of the subterm ρ, we mean its unique evaluation during a reduction of τ (i.e. when free variables in ρ 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 ρ 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 τ evaluates to a non-empty instance.

- For a subterm of the form $\sigma_{p=v}\rho$ we return $\mathsf{Eval}(\rho, S \cup \{(p, v)\})$. For $\sigma_{p_1=p_2}\rho$ we guess a value v from the active domain of the term and return the conjunction of $\mathsf{Eval}(\sigma_{p_1=v}\rho, S)$ and $\mathsf{Eval}(\sigma_{p_2=v}\rho, S)$.
- For a subterm $\rho_1 \mathsf{X} \rho_2$ we calculate the arity of ρ_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 $\mathsf{Eval}(\rho_1, S_1)$ and $\mathsf{Eval}(\rho_2, S - S_1)$.
- For a subterm $\rho_1 \setminus \rho_2$ we return $\mathsf{Eval}(\rho_1, S) \setminus \mathsf{Eval}(\rho_2, S)$.
- For a subterm $\pi_{[l,u]}\rho$ we return

  $\mathsf{Eval}(\rho, \{(c_l, v_l), (c_{l+1}, v_{l+1}), \dots (c_u, v_u)\})$.
- For a subterm $\rho_1 \cup \rho_2$ we return the union of $\mathsf{Eval}(\rho_1, S)$ and $\mathsf{Eval}(\rho_2, S)$.

- For a subterm D_0, where D_0 is a data constant, we simply calculate the projection of $\bigwedge_i \sigma_{c_i=v_i}\mathsf{D}_0$ and return true iff this is nonempty.
- Finally, for a subterm consisting of a relational variable R, we let ρ_1 be the subterm to which R is applied in τ; we return $\mathsf{Eval}(\rho_1, S)$.

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. □

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 τ of degree $k = 1$ we proceed by applying β-reduction to get to a term of degree 0. Both d and m do not change during the reduction process, while the height of the resulting degree 0 term is bounded by $2^{|\tau|}$.

Hence we have:

Proposition 7.6. *Degree* 1 *weakly-typed* HO *terms can be evaluated in* EXPTIME.

7.4 Conclusions

We have studied the general typeability problem for higher-order terms in HO and HOCV. This problem checks if a term is typeable when none or some of its subterms are given specific types.

In the relational case, we have considered a weaker "weakly-typed" version, and shown 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.

With regard to the general typeability problem for XQH terms, we have given preliminary complexity results, which are obtained from those for HOCV terms. In future work, we plan to look at this problem for XQH in more detail. We will also look at a finer type system with *structural types* – e.g., XML schemas. Query and programming languages with support for XML schema types have been studied extensively over the last decade – starting with the language XDuce of Hosoya and Pierce [HP03] and continuing through the language CDuce of Benzaken et al. [BCF03]. The interaction of higher-order functions and structural typing is touched upon in [FCB08], although algorithmic aspects are not pursued in that work.

In future work for relational data, we will consider other extensions of HO 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.

Chapter 8

Higher-Order Query Language Implementation

In the previous chapters, we have considered the complexity in the worst case of the most fundamental problems for higher-order queries over relational data and XML data. In this chapter, we look at the evaluation problem from a practical point of view. This chapter describes a Higher-Order Mapping Evaluation System (HOMES) that integrates querying and query transformation in a single higher-order query language. The system allows users to write queries that integrate and combine query transformations. The power of higher-order functions also allows one to succinctly write complex relational and XML queries. We also show the utility of the system and explain the implementation architecture on top of a relational DBMS and an XML Database engine. We explain optimizations that combine subquery caching techniques from relational/XML databases with sharing detection schemes from functional programming. We give an introduction to graph reduction in λ-calculus, which is used to detect sharing; we will adapt graph reduction in our system.

8.1 Motivation and introduction on graph reduction

Every closed term in our language is expressible in SQL or XQuery, supplemented with recursion in case Inflationary Fixed Point is used. Thus we have a naive method to evaluate higher-order terms: Apply β-reduction until no higher-order abstractions are present, then convert the term to an equivalent SQL query or XQuery expression and evaluate using a standard relational engine or XML Database engine.

In the naive evaluation above, we might repeat many parts of a term when converting to an SQL query or an XQuery expression, especially when there are higher-order

variables which support repetitions of subterms. Many of these copying operations do not need to be carried out during the reduction, since later reductions eliminate them. This is a common problem in the evaluation of functional programs, and we adapt the technique of "graph reduction" to decide which subterms need not be copied.

Graph reduction was first introduced by Wadsworth in his Ph.D. thesis [Wad71]. The thesis presents an algorithm to reduce an input term to a normal form if one exists. The algorithm is the first to use multiple occurrences of variables in optimization of β-reduction. The formal definition of graph reduction is given in Section 8.3. To get the intuition, we consider the following examples.

Example 20. *Let $\tau = (\lambda x.x(xa))((\lambda y.y)b)$ be the term that we need to normalize.*

A naive approach can be $\tau = ((\lambda y.y)b)(((\lambda y.y)b)a) = b(((\lambda y.y)b)a) = b(ba)$. However, we notice that x occurs twice, and both of them will be substituted by $((\lambda y.y)b)$, which is reduced to b. In a graph reduction, this property is used to improve the reduction process.

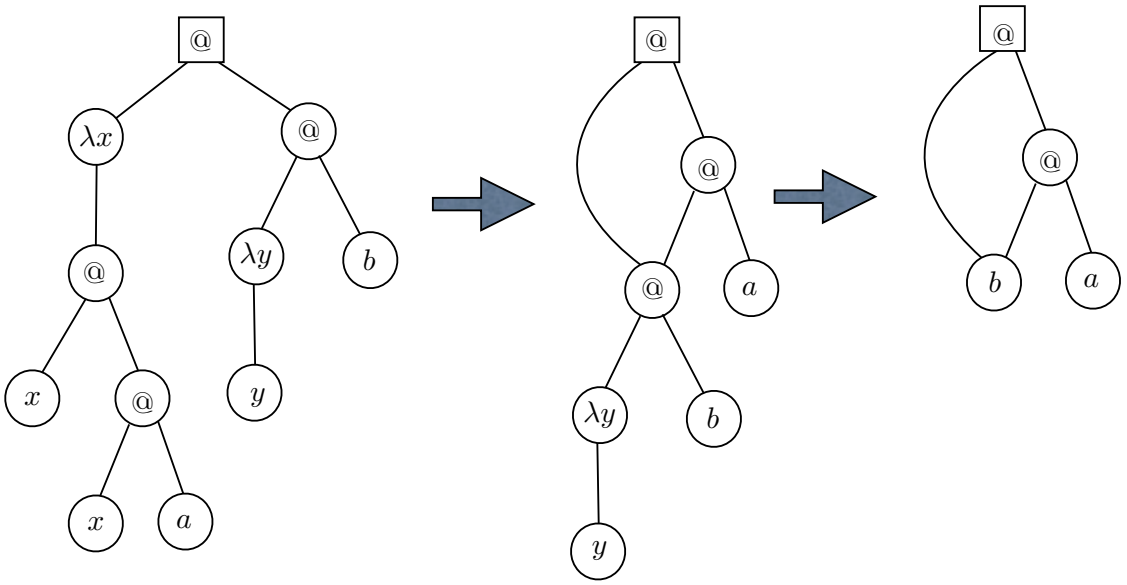


Figure 8.1: Graph reduction with sharing.

However, in another example, the use of sharing is not straightforward.

Example 21. *Let $\tau = (\lambda x.x(xa))(\lambda y.yb)$ be the term that we need to normalize.*

A naive approach can be $\tau = (\lambda y.yb)((\lambda y.yb)a) = ((\lambda y.yb)a)b = (ab)b$. However, we notice that x occurs twice, and both of them will be substituted by $(\lambda y.yb)$, whose variable y is not reducible within this subterm. This example shows us that a graph reduction not only uses sharing subterms but also requires copying.

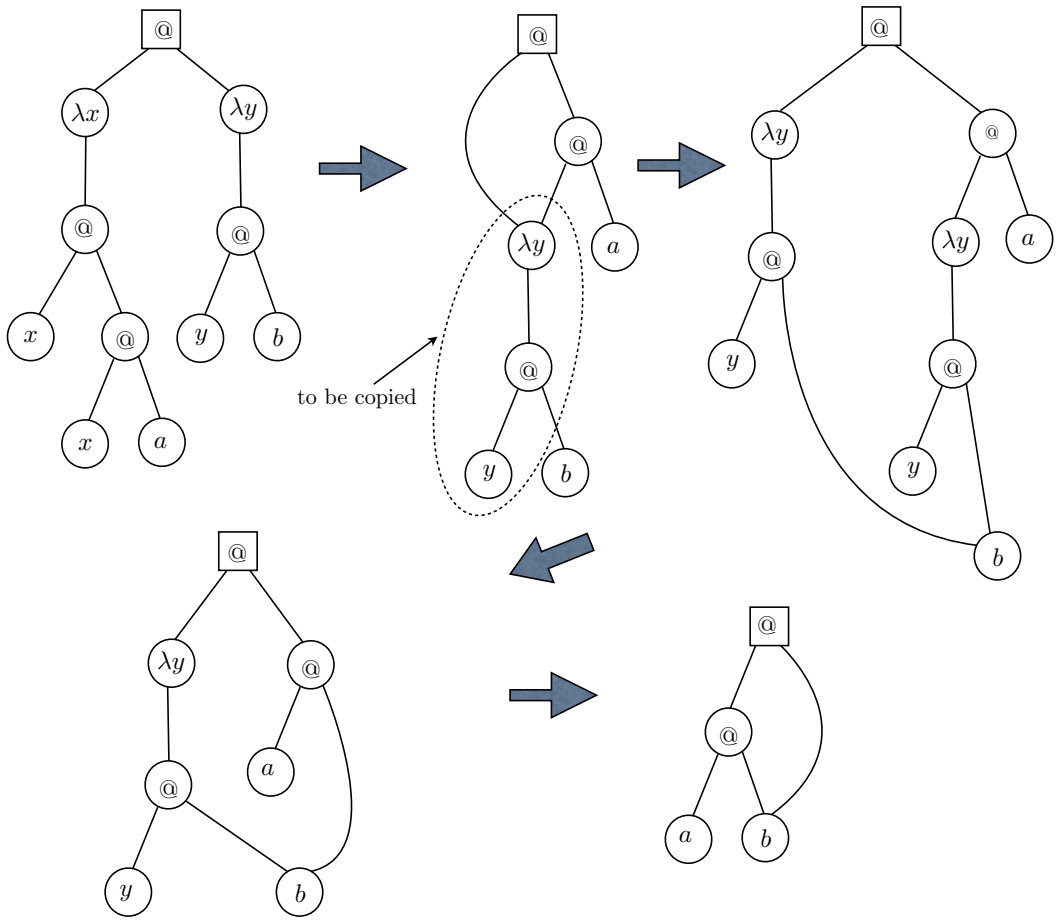
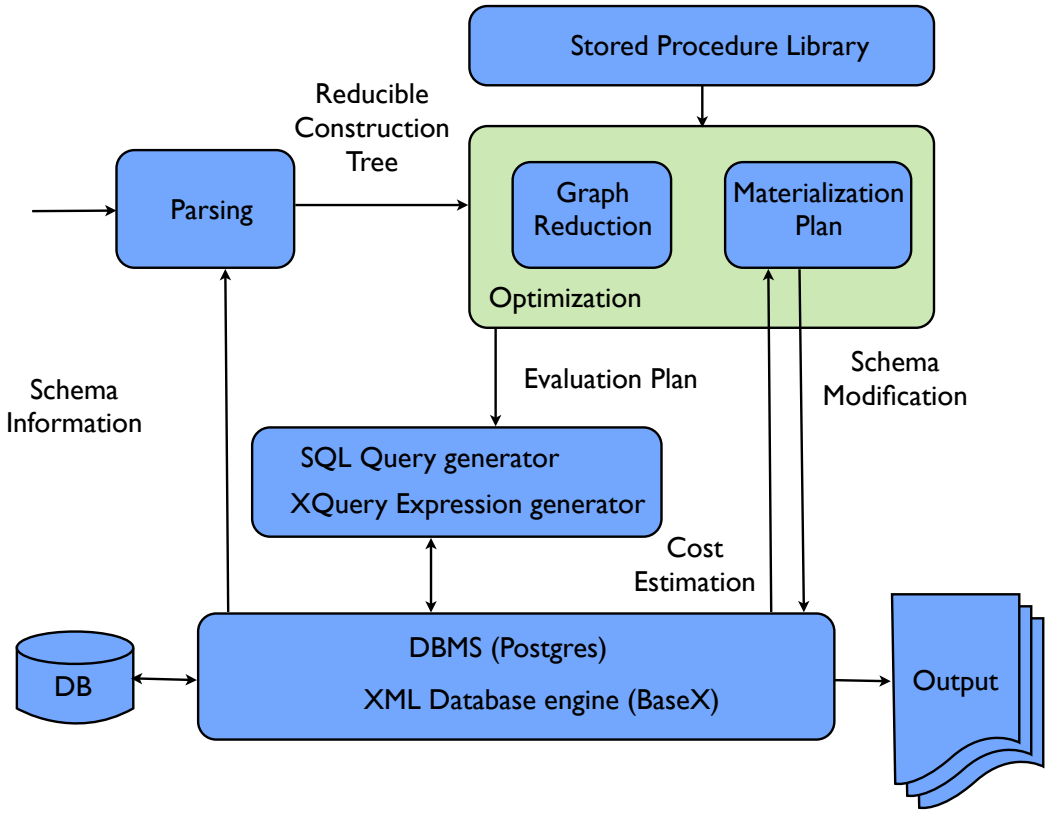


Figure 8.2: Graph reduction with sharing and copying.

Our graph reduction not only reduces λ-variables but also calculates how many times a subtree is called, which can be used to decide if the subtree can be shared. Instead of always copying subtrees, we attempt to share subtrees as much as possible. When the graph reduction process finishes, we have a graph, subgraphs of which contain information about the number of parents.

Note that this is not as simple as detecting common subqueries within a collection of views, since the shared subterm may contain variables of high order, and sharing may not be present in the original term but may emerge in the process of reduction. Sharing subterms is useful even when the shared subterms return higher-order objects; but it is particularly helpful for subterms returning relations, since there we can calculate the result and store it in an auxiliary table. That is, we perform (online) *materialization*, analogous to materialization of views in standard relational query processing.

8.2 System architecture



We will explain the components of the system architecture, which is shown in Figure 8.2 through a running example for the relational database case. Let the input higher-order term be defined as:

$$\tau := \left(\lambda Q.\lambda R_1.\tau_p^2\{\sigma_{b>5}(Q(R_1))\}\right)(\lambda R_2.\sigma_{a=3}(R_2))$$

with τ_p^2 an ordinary conjunctive query checking for the existence of a path of length 2 in such a relation as in Example 4 on Page 6. That is, we first form a query transform that filters the query and then performs a self-join; then we apply the transform to a particular selection query. Given D_0 a database instance, we wish to evaluate $\tau(D_0)$.

The *Parsing* component reads the input in different forms from users, parses the input query, and validates its type; it produces an internal representation, a *construction tree*, represented in the upper part of Figure 8.3. In the construction tree, Relational Algebra operators are employed to represent the term. For example, τ_p^2 is represented as $\lambda R.\pi_{a,b}(\rho_{b,c}(R) \bowtie \rho_{a,c}(R))$.

The construction tree in Figure 8.3 is then input to the *Optimization* component, which also takes information from the *Stored Procedure Library* to evaluate higher-order constants in the term. The library contains processing methods of higher-order

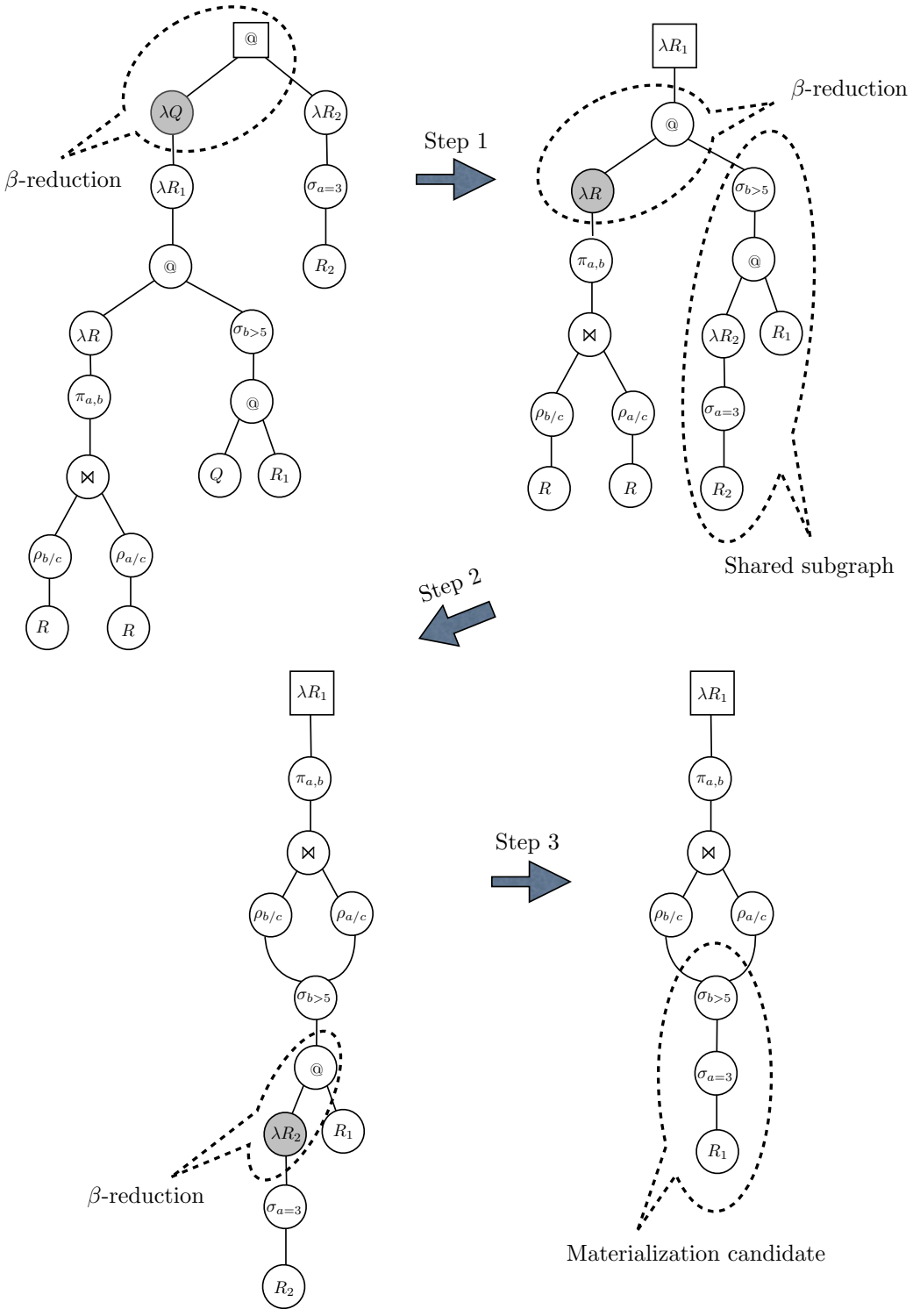


Figure 8.3: The construction tree and the graph reduction stage of τ.

operators, which are either built-in or user defined. Examples of built-in operators currently supported are Inflationary Fixed Point (`ifp`) and Query Rewriting (RW).

The output of the optimization is an evaluation plan, which contains information about the materialization tables and the order to process the subterms of the input term. The evaluation plan is represented by a set of equations of which one side is a table name, and the other side is the equivalent subterm. The Optimization component will be explained in detail in Section 8.3.

The SQL Query Generator - XQuery Expression Generator component implements the evaluation plan, generating queries at runtime to a Relational Database Management System and an XML Database Engine – in our implementation we use PostgreSQL 8.4 and BaseX 7.0.2.

8.3 Optimization

There are thus two stages in the optimization: Graph reduction and generation of an evaluation plan.

Stage 1 (Graph reduction): Graph reduction is based on a directed acyclic graph, called a "reduction graph", which is similar to the definition in [Wad71], but has some changes to fit the other parts of the thesis.

Definition 15 (Reduction Graph)**.** *A reduction graph is an acyclic directed graph (DAG) that consists of a set of nodes $\mathcal{X}$ and a set of directed edges $\mathcal{E}$. The nodes in $\mathcal{X}$ are labeled with: @ (called application nodes, or @ nodes), λR with R a variable name (called λ nodes), and names of variables and constants.*

Reduction graphs have the following properties:

- *λ nodes have one child.*
- *Application nodes have two children*
- *Nodes labeled with names of variables and order 0 constants are leaves.*

In Figure 8.1, Figure 8.2, and Figure 8.3, we depict a number of reduction graphs; the roots are put inside squares and the other nodes are put inside circles. Note that a construction tree is a special kind of reduction graph.

Linear conversion from a reduction graph to a term.
Given a reduction graph, we use the following rules to convert it to a λ expression:

- If the graph is a node N, which is a leaf, then the expression is the label of that node.
- If the graph has a root @ with the left subterm ξ_1 and the right subterm ξ_2, which are respectively transformable to τ_1 and τ_2, then the expression is $\tau_1(\tau_2)$.
- If the graph has a root λx linked to a subgraph ξ_1, which is transformable to a term τ_1, then the expression is $\lambda x.\tau_1$.

Given a higher-order term as the input, we build its construction tree in Algorithm 1 on Page 32.

We use a variant of the graph reduction algorithm described in [Wad71, AG98] to reduce variables. The normalization is described by the method *normalization* in Algorithm 4, where *root* is a function that returns the root of a reduction graph.

Algorithm 4 *normalization*: Normalization of a higher-order term

Input: The construction tree (initial reduction graph) G of the input term ρ
Output: A directed acyclic graph representing the normalized term

1: **while** G has variables **do**
2: $reNode := root(G)$
3: $rightNode := reNode.rightChild()$
4: $leftNode := reNode.leftChild()$
5: **if** $((reNode = @)$ **and** $(leftNode = \lambda X)$ **and** X is of highest order and the order of variables in the right tree of @ is less than the order of $X)$ **then**
6: **if** ($reNode$ has a single parent) **then**
7: $betaReduction(leftNode, varName, rightNode)$
8: $reNode := leftNode.leftChild()$
9: {Remove @ and λX from G}
10: **else**
11: **for** (p a parent of $reNode$) **do**
12: Let $z_p = copy(reNode, varName)$
13: Replace the edge from p to $reNode$ by an edge from p to z_p
14: **end for**
15: **end if**
16: **else**
17: G_{left} := maximal subgraph of G with root $leftNode$
18: G_{right} := maximal subgraph of G with root $rightNode$
19: $normalization(G_{left})$
20: $normalization(G_{right})$
21: **end if**
22: **end while**
23: **return** G

The methods *betaReduction* and *copy* in *normalization* are respectively described in Algorithm 5 and Algorithm 6. The method *betaReduction* implements a β-reduction using sharing, such as in Example 20. Whereas, the method *copy* makes a copy of a subgraph to prepare for β-reduction of variables in shared subgraphs, e.g., step 2 in Example 21.

Algorithm 5 *betaReduction*: Using sharing for β-reduction a variable in the graph

Input: Root *root* of a graph, the variable label *varLabel*, root *subNode* of another graph

Result: Occurrences of *varLabel* in the graph with root *root* are reduced. All the edges to occurrences of *varLabel* are moved to the root *subNode*.

```
1: if (root ≠ null) then
2:   if (label(root.leftChild()) = varLabel) then
3:     root.leftChild := subNode
4:   else
5:     betaReduction(root.leftChild(), varLabel, subNode)
6:   end if
7:   if (label(root.rightChild()) = varLabel) then
8:     root.rightChild := subNode
9:   else
10:    betaReduction(root.rightChild(), varLabel, subNode)
11:  end if
12: end if
```

Theorem 8.1. *The normalization procedure described in Algorithm 4 terminates and produces a term in normal form with the same semantics.*

Proof. First, we show that the algorithm terminates and produces a normalization of the input term. Let k be the highest order of variables at an arbitrary step and the term is still reducible by β-reduction.

Line (5) in Algorithm 4 guarantees that variables in the DAG with root $rightNode$ of the call to $betaReduction(leftNode, varName, rightNode)$ in line (7) does not have variables of order k. We show that this condition can be implemented as follows. Given $\mathcal{N}^k$ the set nodes of the form λX with X an order k variable, we always can find a node n in $\mathcal{N}^k$ that is furthest from the root. Since k is the highest order of variables at this step, node n is reducible by β-reduction; otherwise, there must exist a variable of order greater than k.

Thus, the variables in the DAG that will be shared at this step are of order less than k. This means that the *betaReduction* algorithm does not put variables of order k into shared DAGs, i.e these variables will not be copied. Therefore, after a finite

Algorithm 6 *copy*: Creating a partial copy of a subgraph

Input: The root $reNode$ of a subgraph and a variable name $varLabel$
Output: The root of a subgraph

1: Find $subGraph$ the smallest subgraph containing all the paths from $reNode$ to leaves with label $varLabel$
2: Initialize $copyGraph := \emptyset$
3: **for** Each edge e from node n to node m in the graph rooted $reNode$ such that $n \in subGraph$ **do**
4: **if** n was not copied to $copyGraph$ **then**
5: Make a copy n' of n and add to $copyGraph$
6: **end if**
7: **if** $m \in subGraph$ **and** m was not copied to $copyGraph$ **then**
8: Make a copy m' of m and add to $copyGraph$
9: **end if**
10: **if** $m \notin subGraph$ **then**
11: $m' := m$
12: **end if**
13: Add to $copyGraph$ the edge from n' to m'
14: **end for**
15: **return** $root(copyGraph)$

number of calls to $betaReduction(leftNode, varName, rightNode)$, all the variables of order k will be reduced. Step by step, all variables of order $k-1$ are reduced, then variables of order $k-2$ are reduced, ... After a finite number of steps, all variables that can be reduced by β-reduction are substituted, i.e. the resulting DAG represents a term in normal form.

Second, we show that the normalized term represented by the resulted DAG has the same semantics as the one represented by the input DAG. This is shown by case analysis using the linear transformation from a reduction graph to a higher-order term. Notice that the *betaReduction* and *copy* algorithms do not affect the linear transformation. Thus, at each step in *normalization*, the term obtained from G by applying the linear transformation has the same semantics as the one obtained through β-reduction. □

Our running example τ from Section 8.2 shows the process at its simplest. From the construction tree in the upper part of Figure 8.3, we reduce to the DAG shown at the bottom of the figure after three steps. In the first step, variable Q is substituted by a DAG representing an order 1 subterm. Since there is only one occurrence of Q in the first DAG, there is no sharing. In step 2, variable R is reduced by replacing its occurrences by pointers to the substituted subgraph, thus introducing sharing. The

shared subgraph contains a variable R_2 and it can be reduced. The reduction of the shared subgraph produces multiple simplifications in the construction tree. At the end, the shared subterm is a selection over R_1.

The output of this stage is a set of candidates for materialization. In our running example, the shared subgraph of the last graph is a candidate for materialization because it is called twice.

Stage 2 (Generation of an evaluation plan): Based on the reduced graph, we can decide which subgraphs should be materialized. Our optimizer produces a materialization plan for the graph. The decisions about materialization are based on the graph and a cost function, roughly similar to the strategies used to decide materialization in a cost-based way in XML query processing over relational stores (see, e.g., [BCF+02]).

Given a reduction graph G without @ nodes, we use the function $graphToQuery$ to convert to an SQL query or an XQuery expression $Q = graphToQuery(G)$. In the following, we assume that given a graph G, there exist functions $C_{eval}(G)$ and $\mathcal{S}(G)$ that can estimate the evaluation cost and the output size, respectively. From the graph, we also find the number of times a query is used by another query.

Given a subgraph g_1 of G such that all the leaves of g_1 are also the leaves of G, the materialization cost of the graph g_1 is defined as:

$$C_{mat}(g_1) = C_{eval}(g_1) + \omega \cdot \mathcal{S}(g_1)$$

with ω a parameter obtained from experiments.

Let $C_{eval}(G \mid g_1, g_2, \ldots, g_n)$ denote the cost of evaluating G when $g_1, g_2, \ldots, g_n$ are materialized. The benefit of materialization of g_1 when $g_2, \ldots, g_n$ have been materialized is defined as:

$$gain(G \mid g_1) = C_{eval}(G \mid g_2, \ldots, g_n) - C_{eval}(G \mid g_1, g_2, \ldots, g_n) - C_{mat}(g_1)$$

Note that only $g_2, \ldots g_n$ that are subgraphs of g_1 will affect $gain(G \mid g_1)$.

From the definition of the materialization benefit above, we implement a dynamic programming algorithm described in Algorithm 7. Intuitively, the algorithm first puts all the nodes in the input graph G into a list L of potential nodes for materialization. Then it travels bottom-up on G and removes from L the nodes that do not benefit the evaluation of G.

Algorithm 7 uses a general C_{eval}. We now look at what happens when a particular formula for $C_{eval}(G \mid g_1)$ is used. Given g_1 a materialized subgraph of G above,

Algorithm 7 *matDynamic* Dynamic materialization for evaluation for higher-order queries

Input: The root *rootG* of a reduction graph G
Result: The set of nodes in G for materialization.

```
1: Initialize the materialization list L as the set of nodes in G
2: Mark all the leaves of G as checked and the other nodes in G as unchecked
3: while L is changed do
4:   if root(g) ∈ L and root(g) is unchecked and the other nodes in g are checked
     then
5:     mark root(g) as checked
6:     if gain(G | g) ≤ 0 then
7:       Remove root(g) from L.
8:     end if
9:   end if
10: end while
11: return L
```

consider the evaluation cost given by:

$$C_{eval}(G \mid g_1) = C_{eval}(G) - k_1 \cdot C_{eval}(g_1)$$

where k_1 denotes the number of parents of the root of g_1. Then the benefit of materializing a subgraph g_1 is equal to:

$$\begin{aligned} gain(G \mid g_1) &= C_{eval}(G) - (C_{eval}(G) - k_1 \cdot C_{eval}(g_1)) - C_{mat}(g_1) \\ &= (k_1 - 1) \cdot C_{eval}(g_1) - \omega \cdot \mathcal{S}(g_1) \end{aligned}$$

Proposition 8.2. *The benefit to the evaluation of G when a subgraph of G is materialized is independent on the materialization of other subqueries.*

Proof. Given that subterms $g_1, \ldots, g_n$ of G have been materialized, we calculate extra gain when another subterm g_{n+1} of G is materialized. Assuming that g_{n+1} is used by $G, g_1, \ldots, g_i$ respectively $k_0, k_1, \ldots, k_i$ times, then the gain of materializing g_{n+1} when the cost of evaluating $G, g_1, \ldots, g_i$ respectively is reduced by $(k_0 \cdot C_{eval}(g_{n+1}), \ldots, k_i \cdot C_{eval}(g_{n+1})$. Thus, we have:

$$gain(G \mid g_1, \ldots, g_{n+1}) = gain(G \mid g_1, \ldots, g_n) - C_{mat}(g_{n+1}) + \textstyle\sum_{j \in \{0,\ldots,i\}} k_j \cdot C_{eval}(g_{n+1})$$

$$gain(G \mid g_1, \ldots, g_{n+1}) - gain(G \mid g_1, \ldots, g_n) = -C_{mat}(g_{n+1}) + \textstyle\sum_{j \in \{0,\ldots,i\}} k_j \cdot C_{eval}(g_{n+1})$$

$$gain(G \mid g_1, \ldots, g_{n+1}) - gain(G \mid g_1, \ldots, g_n) = -\omega \cdot \mathcal{S}(G) + (K - 1) \cdot C_{eval}(g_{n+1})$$

with $K = \sum_{j \in \{0,\ldots,i\}} k_j$, which is equal to the number of parents of the root of the DAG corresponding to g_{n+1}.

This means that when we materialize a subterm of G, the gain to the evaluation of G is independent of the materialization of other subqueries. □

Based on Proposition 8.2, we describe in Algorithm 8 a simpler algorithm for materialization, which materializes every subgraph G' of G such that $gain(G \mid G') > 0$.

Algorithm 8 *matGraph* Materialization for evaluation higher-order queries

Input: The root *rootG* of a reduction graph G
Result: The set of nodes in G for materialization.

```
1: Initialize the materialization list L = ∅
2: for G' a subgraph of G do
3:    if (gain(G | G') > 0 ) then
4:       Add the root of G' to L
5:    end if
6: end for
7: return L
```

The materialization plan contains a set of nodes, which are roots of subgraphs to be materialized. Each subgraph is materialized into a relation, which links to the root of that subgraph by an equation. In our example, a possible materialization plan is:

$R_0 \Leftarrow$ SELECT $*$ FROM R_1 WHERE $a = 3$ AND $b > 5$

$Output \Leftarrow$ SELECT $R_0^1.a, R_0^2.b$
FROM R_0 AS R_0^1, R_0 AS R_0^2 WHERE $R_0^1.b = R_0^2.a$

where R_0 is a new table name for materialization.

8.4 The differences between relational implementation and XML implementation

This section describes the main differences between implementing an HO evaluation system and implementing an XQH evaluation system.

Input syntax.
For the relational case, we use a variation of the syntax of the higher-order language

HO defined in Figure 4.1 on Page 25. The input syntax in the implementation uses $s(c), p(A), j, r(a,b), u$ and m to denote the Relational Algebra operators $\sigma_c, \pi_A, \bowtie$, $\rho_{a/b}, \cup$ and $\setminus$, respectively. It also uses $\setminus$ to denote λ. When querying XML data one cannot use monad algebra, since the data models do not match. Thus, in the XML case, the implementation only allows query constants as Core XQuery expressions defined in [Koc06]. The input syntax of the implementation for the XML case is the syntax of the higher-order language XQH, which is defined in Figure 4.3 on Page 38.

Construction trees.

The nodes in the construction tree for the relational case built by applying Algorithm 1 are λ nodes, @ nodes, variable nodes, and constant nodes. The constant nodes are labelled with relation names and Relational Algebra operators. For the case of higher-order queries over Core XQuery, the Relational Algebra operators in the construction tree are replaced by higher-order constants of order 1 or 2 (XQH query constants) defined by users and following the syntax of XQuery. In the implementation, we allow users to define these constants as normalized terms of order at most 2 that have at most two λ variables. The tree is inductively built as in Algorithm 9.

Algorithm 9 Building the construction tree of an XQH term

Input: An XQH higher-order term
Output: A construction tree

1: The root of the tree is the outermost operator.
2: The construction tree of an order 0 constant (resp., a variable) is a single leaf, and labelled by the constant's name (resp., the variable's name).
3: The construction tree of a constant with one λ variable $\lambda x.\tau_c$ has the root labelled λx, which has a child node labelled τ_c. The node labelled τ_c has a single child, which is a leaf, and labelled x.
4: The construction tree of a constant with two λ variables $\lambda x.\lambda y.\tau_c$ has the root labelled λx, which has a child node labelled τ_c. The node labelled τ_c has two children, which are leaves, and labelled x and y.
5: The construction tree of an abstraction $\lambda x.s$ consists of a node labelled λx with a single subtree, which is the construction tree of s.
5: 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.

Materialization decisions.

In the relational case, we use the cost estimation facility provided by the DBMS

to support the decision for materialization. However, this facility does not exist in XML database engines that we have tested. Thus, for the XML case, we use a set of rules to decide if a DAG should be materialized. For example, the rules may be materialized based on the number of the roots of the DAG and the selectivity of the query obtained from the DAG. In the current implementation, we materialize all the roots of the DAGs that satisfy: (i) The roots have more than one parent, and (ii) the XQuery expressions obtained from DAGs contain equivalence checking in the WHERE clause.

Materialization methods.
Materializing a DAG in the relational case is straightforward by storing the output of the SQL query obtained from the DAG into a database table. For the XML case, the output of the XML query can be either a node or a sequence of nodes. In the latter case, we need to create a dummy node before putting it into an XML document in the XML database engine. The use of a materialized DAG needs to take these dummy nodes into account. As our higher-order language for XML does not support upward navigation axes, materializing a DAG is not affected by the other nodes outside the DAG. Additionally, the higher-order language does support node ID comparison; otherwise, the equality and inequality between the nodes do not hold after adding the dummy nodes. For example, we consider the following query that can be affected by dummy nodes from materialization:

```
let $Q := 'xmark116mb.xml'/site/people/
for $x in $Q, $y in $Q
where $x <> $y
return <a>True</a>
```

Assuming $\$Q$ is not empty, if we materialize $\$Q$ and add a dummy node, then the query above always return $\langle a \rangle True \langle /a \rangle$; whereas, the original query can return an empty output.

8.5 Empirical evaluation

The implementation of HOMES shows the following two main aspects. First, we show the usefulness of integrating higher-order queries into a relational DBMS or an XML Database Engine. The integration allows users to create and reuse higher-order queries, including variables representing queries and query transformations. Secondly, we show how to extend materialization as an optimization technique to higher-order

query languages. We show how our techniques yield acceptable running time even for complex queries for the cases of relational data and XML data.

Experiments over relational data.
First, we consider the performance of our implementation on a set of queries based on the TPC-H schema [Cou]. Then we consider its performance on a set of data containing flight routes [Wah].

The test database from the TPC-H schema contains the following tables: *customer*, *lineitem*, *nation*, *orders*, *part*, *partsupp*, *region*, and *supplier*. TCP-H gives default sizes for each table, and in our experiments instances are generated by uniformly multiplying the default size by a scaling factor. The experiments study some higher-order queries on the table *parsupp*, which has five columns: *partkey* (integer), *suppkey* (integer), *ps_availqty* (integer), *ps_supplycost* (numeric), and *ps_comment* (character varying).

Table 8.1 presents information on evaluation of queries containing joins over instances of different scales, denoted k, on an Intel ®Core i3 2.27 GHz machine with 4 GB RAM. The last four rows in the table show the evaluation time (with second (s) the default unit) for the queries above using two different evaluation plans: Naive and Graph reduction, over the database with default size ($k = 1$) and a scaling factor ($k = 5$). The evaluation time of the queries in Table 8.1 shows the improvement due to graph reduction and sharing, especially for the queries containing multiple joins.

	Query 1	Query 2	Query 3	Query 4	Query 5	Query 6
Query description	1 join	4 joins	8 joins	16 joins	with `ifp`	with RW
Improved, $k = 1$	2.4 s	1.1 s	5.7 s	6.7 s	3.5 s	1.3 s
Naive, $k = 1$	3.0 s	1.2 s	164.8 s	18930 s	11.6 s	7.6 s
Improved, $k = 5$	5.3 s	3.8 s	9.3 s	12.3 s	10.2 s	4.7 s
Naive, $k = 5$	6.5 s	143.9 s	1137 s	$>$ 12 h	92.3 s	62.2 s

Table 8.1: Empirical evaluation results for the relational data case.

We give details of the queries in Table 8.1 (in the query description field). In these queries, we use a predefined type:

$$\mathcal{T}_0 = (partkey, suppkey, ps_availqty, ps_supplycost, ps_comment)$$

We also use a relational constant D_0 mapping to the table *partsupp*, three relational variables R_1, R_2 and R_3 of type $\mathcal{T}_0$, and two query variables Q_1 and Q_2 of type $\mathcal{T}_0 \to \mathcal{T}_0$.

Below are the higher-order terms described in the table.

Query 1: One natural join

```
( \R1 p(partkey,suppkey) ( ( r(suppkey,c) R1 ) j ( r(partkey,c) R1 ) ) )
( s(suppkey<200)  D0 )
```

Query 2: Two nestings of a query variable – 4 joins

```
( ( \Q ( \R1  Q ( Q ( s(partkey<10)  R1  ) ) ) )
( \R2 ( r(suppkey,c) R2 ) j ( r(partkey,c) R2 ) ) )
( s(suppkey<200) D0 )
```

Query 3: Three nestings of a query variable – 8 joins

```
( ( \Q1 ( \R1 Q1 ( Q1  ( Q1 ( s(partkey<100)  R1  ) ) ) ) )
( \R2  p(partkey,suppkey,ps_availqty,ps_supplycost,ps_comment)
( ( r(suppkey,c) R2 ) j ( r(partkey,c) R2 ) ) )
) ( s(suppkey<200)  D0  )
```

Query 4: Two nestings of two query variables – 16 joins

```
( ( \Q1  \R1 ( Q1 ( Q1 ( R1 ) ) ) )
( ( \Q2  \R2 ( Q2 ( Q2 ( R2 ) ) ) )
( \R3  p(partkey,suppkey,ps_availqty,ps_supplycost,ps_comment)
( ( r(suppkey,c) s(partkey<10) R3 ) j ( r(partkey,c) s(partkey>5) R3 ) ) ) ) )
( s(suppkey<200) D0 )
```

The following two queries demonstrate how the Fixed Point and Query Rewriting operators are used in our higher-order queries. In these two queries, R is of type $(partkey, suppkey)$.

Query 5: Using the inflationary fixed point operator **IFP** inside a higher-order term

```
(s(suppkey<50) (p(partkey,suppkey) D0)) u
IFP(p(partkey,suppkey) (s(suppkey<50) D0),
(\R p(partkey,suppkey) ((r(suppkey,c) R) j (r(partkey,c) (s(suppkey<10) D0)))))
```

Query 6: Using the rewriting operator as a higher-order constant **RW** inside a term

```
Pre-defined terms:
H(x,t) <- X1(x,y) , X1(y,z) , X1(z,t)
V1(1u,1v) <- X1(1u,1t) , X1(1t,1v)
V2(2u,2v) <- X1(2u,2v)
V3(3u,3u) <- X1(3u,3)

Main term:
( s(partkey<100) RW(H,V1,V2) ) u ( s(suppkey<50) ( p(partkey,suppkey) D0 ) )
```

In the queries above, we use the join condition $partkey = suppkey$, which is not natural. Now, we consider three queries over a table *route* containing six columns: *depart* (character varying), *dest* (character varying), *time* (integer), *period* (character varying), *alt* (character varying), and *type* (character varying). Our higher-order queries will create multiple joins of *route*. These queries output pairs of airports between which there is a flight route with a specific number of transits. A relational constant $D1$ is defined as *route*. Table 8.2 presents the evaluation results of Queries 7, 8, and 9.

	Query 7	Query 8	Query 9
Query description	1 join	4 joins	8 joins
Improved	2.9 s	1.5 s	46.2 s
Naive	3.1 s	611.9 s	> 6 h

Table 8.2: Empirical evaluation results for HO queries over the table *route*.

Below are the higher-order queries described in Table 8.2. In these queries, R is of type $(depart, dest)$, R_1 and R_2 are of type $(depart, dest, time, period, alt, type)$, Q is of type $(depart, dest, time, period, alt, type) \rightarrow (depart, dest, time, period, alt, type)$

Query 7: One natural join

```
( \R  p(depart,dest) ( ( r(dest,c) R ) j ( r(depart,c) R ) ) )
( p(depart,dest) ( s(time<1200)  D1 )  )
```

Query 8: Two nestings of a query variable – 4 joins

```
( ( \Q ( \R1 Q  ( Q (  R1  ) ) ) )
( \R2 ( ( r(dest,c) R2 ) j ( r(depart,c) R2 ) ) ) )
( s(time<1200) D1 )
```

Query 9: Three nestings of a query variable – 8 joins

```
( ( \Q ( \R1 Q ( Q  ( Q ( R1 ) ) ) ) )
( \R2 ( ( r(dest,c) R2 ) j ( r(depart,c) R2 ) ) ) )
( s(time<1000) D1 )
```

Overall, we have seen significant performance improvement by applying the graph reduction plan, especially when the materialized tables are accessed many times.

Experiments over XML data.

Our experimental work with the XML implementation is limited to testing basic

functionality and a few observations related to performance. We evaluate the performance of the XML data part of the implementation on a set of queries based on the XMark benchmark [BCF+, SWK+02]. We will use the descriptions of the queries as presented in [SWK+02]. The test database is generated using the data generator *xmlgen* of the benchmark. The default size of the XML files generated by *xmlgen* is 116 MB, which can be changed by a scaling factor.

XQH **Query 1:** List the names of all persons and the number of items they bought.

```
CONST:
D1 : xmark116mb.xml/site/people/person
D2 : xmark116mb.xml/site/closed_auctions/closed_auction
E1 : \R1 \R2 for $p in R1
          let $a := for $t in R2
          where $p/@id = $t/buyer/@person
          return $t
     return <item person="{$p/name/text()}">{count($a)}</item>
TERM:
( E1 D1 ) D2
```

XQH **Query 2:** For each person, list the number of items whose price does not exceed 0.02% of the person's income.

```
CONST:
D1 : xmark116mb.xml/site/people/person
D2 : xmark116mb.xml/site/open_auctions/open_auction/initial
E1 : \R1 \R2 for $p in R1
          let $a := for $t in R2
          where $p/profile/@income > 5000 * exactly-one($t/text())
          return $t
     return <items name="{$p/name/text()}">{count($a)}</items>
TERM:
( E1 D1 ) D2
```

XQH **Query 3:** For each richer-than-average person, list the number of items currently on sale whose price does not exceed 0.02% of the person's income.

```
CONST:
D1 : xmark116mb.xml/site/people/person
D2 : xmark116mb.xml/site/open_auctions/open_auction/initial
E1 : \R1 \R2 for $p in R1
          let $a := for $t in R2
          where $p/profile/@income > 5000 * exactly-one($t/text())
          return $t
     where $p/profile/@income > 50000
     return <items person="{$p/profile/@income}">{count($a)}</items>
TERM:
( E1 D1 ) D2
```

XQH **Query 4:** List the names and descriptions of the items that are registered in Australia.

```
CONST:
D1 : xmark116mb.xml/site/regions/australia/item
F1 : \R1 for $i in R1
     return <item name="{$i/name/text()}">{$i/description}</item>
TERM:
F1 D1
```

The queries above, which run XQuery on XML data, show the correctness of the implementation over ordinary queries.

In the following queries, we consider the performance of the implementation on 28 MB XML data containing flight routes [Wah]. As in the relational case, we can write realistic queries with a big number of joins over this XML data. However, due to the limit in processing queries with large joins of the XML database engine that the implementation is based on, the current implementation does not allow us to test higher-order queries containing as many joins as those in the relational case. Thus we only give experiment results for the queries with at most four joins.

XQH **Query 5:** Join between direct flight routes to find routes with one transit.

```
CONST:
D1 : graph4.xml//route
E1 : \R1 \R2 for $i in R1//connect, $j in R2//connect
     where substring($i/text(),6,4) = substring($j/text(),1,4)
     return <rt>{ $i/text(),"-",$j/text() }</rt>

TERM:
( E1 D1 ) D1
```

XQH **Query 6:** Join between direct flight routes to find routes with three transits.

```
CONST:
D1 : graph1140.xml//route
F1 : \R <connects>{ for $i in R//connect, $j in R//connect
     where substring($i/text(),6,4) = substring($j/text(),1,4)
     return <connect>{ fn:concat(substring($i/text(),1,4),"-",substring($j/text(),6,4))
                     }</connect> }</connects>

TERM:
F1 ( F1 D1 )
```

Our experiments with XQH Query 6 have shown the improvement of the system when it uses graph reduction, which is described in Section 8.3. The evaluation time using a naive plan is 60 seconds; whereas, the evaluation using graph reduction takes only 21 seconds.

8.6 Prototype details

To highlight the aspects in the evaluation section above, users are guided through query development and evaluation using the system's GUI. They can begin with a number of pre-rewritten higher-order queries, applying them to several sample databases. Our higher-order constant Query Rewriting is implemented based on the *bucket algorithm* presented in [LRO96a, LRO96b]. Once the users are familiar with the syntax, they can modify pre-existing queries or create new ones from scratch. The input syntax that the prototype currently supports is higher-order queries over Relational Algebra and Core XQuery. The evaluation of the queries can be done live, showing both the output and the execution time for several variants of the evaluation algorithm.

Figure 8.4 illustrates the GUI for relational data. The top-left of the GUI is the input area, where a user can input a higher-order query or load it from a file. The top-right area and bottom-right area contain pre-defined types and pre-defined terms, respectively. The user also can use these areas to define their own types and terms as macros, which are used in the input area. The bottom-left area contains the output obtained after clicking the *Evaluate* button. Clicking the *Show* button will open a new dialog that contains output tuples.

Two buttons *Graphreduction* and *Naivereduction* provide facilities to illustrate the graph reduction and materialization process in action, e.g., Figure 8.5. In this figure, the λ nodes *lambda* Q, *lambda* R_1, and *lambda* R that are reduced by β-reduction at each step are marked in gray. In the final DAG, the potential nodes for materialization are highlighted.

8.7 Conclusions

We have developed a prototype implementation to show the practical use of higher-order queries for relational and XML databases. These higher-order queries can be used as an extension of relational database queries and XQuery expressions. This extension is important especially for the relational case, because reusing queries, including order 0 ones, is not supported in the SQL syntax itself. For the case of XQuery, the *let* operator is already a tool for reusing expressions of order 0. Moreover, BaseX and some other XML database engines also support XQuery functionals, which are order 1 expressions in our higher-order language, as in Example 6 in Section 1.1 of Chapter 1. However, there has been no research work on optimizing order 1

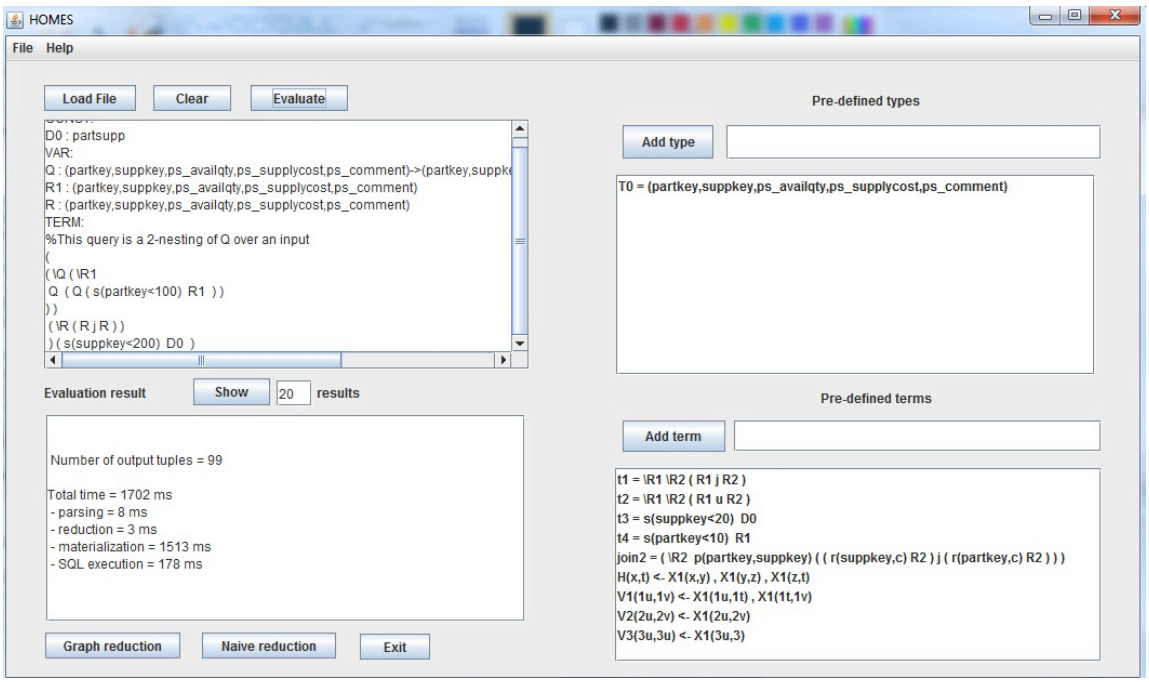


Figure 8.4: A screen-shot of the prototype.

functionals. Our system supports reuse of terms of arbitrary order and provides optimization techniques for evaluating higher-order terms. Additionally, we believe that the simple syntax used in our system is easier for users to experiment with.

There are a number of directions to improve the current implementation. Here we only state a few of them; more are given in the Conclusion chapter. The materialization for the XML case is not as efficient as that for the relational case because we do not have facilities to estimate the cost of evaluating and storing an XQuery expression. We hope this problem will be solved when XML database engines support these facilities. The future implementation should support higher-order queries on the full XQuery language because the current implementation is still limited to Core XQuery. We also plan to extend the prototype to include higher-order queries for SQL.

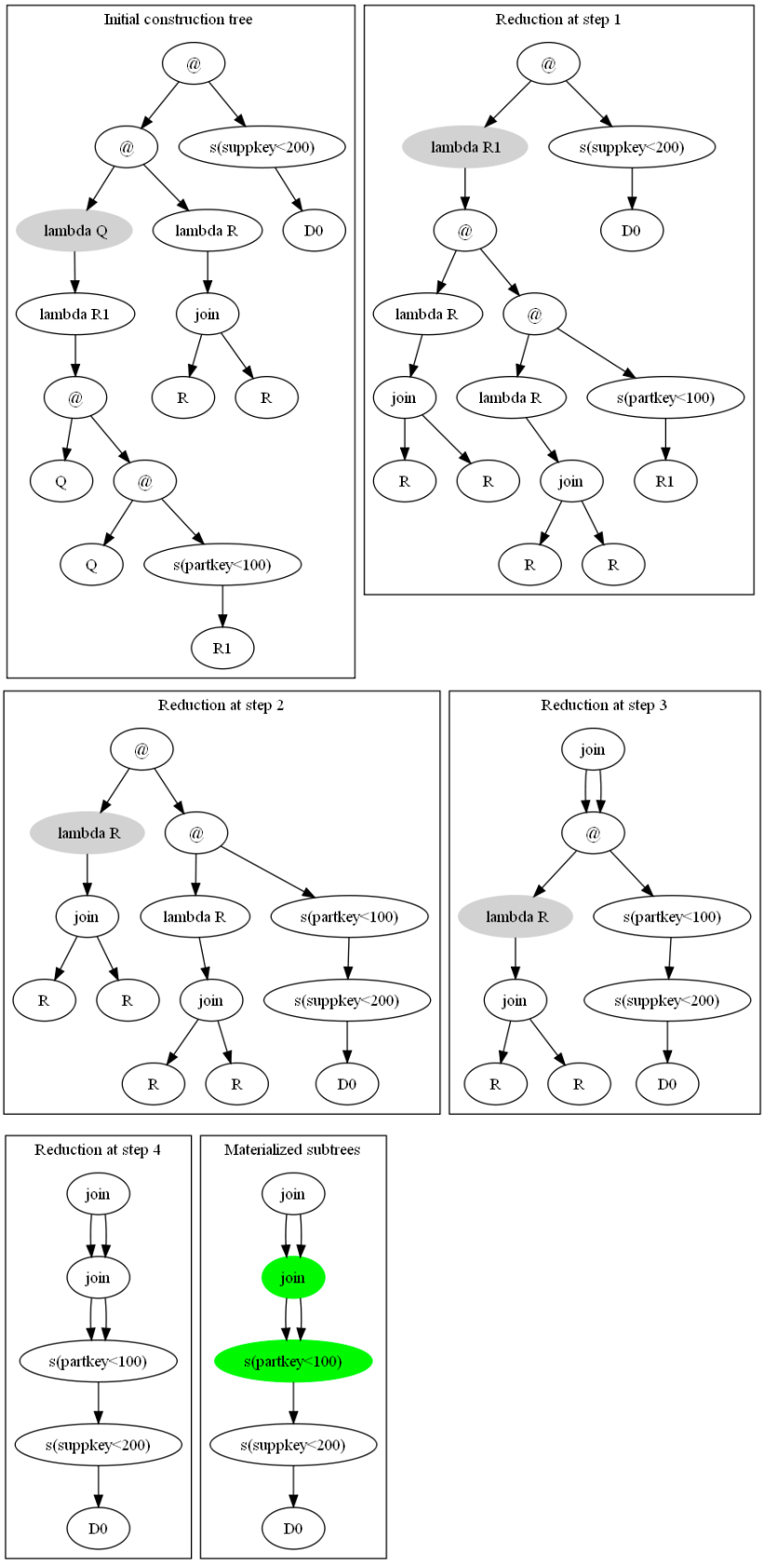


Figure 8.5: A screen-shot of graph reduction generated in the prototype.

Chapter 9

Conclusions and Future Work

This chapter concludes the thesis and suggests future directions.

9.1 Conclusions

The thesis studies the combination of λ-calculus and database queries to create higher-order query languages over relational databases, XML databases, and complex values. In addition to elements of λ-calculus, these languages contain constants relating to data manipulation. For the case of relational databases, the constants consist of database instances and Relational Algebra operators. The higher-order language for XML databases contains XQuery expressions and XML documents as constants. For the case of complex values, the constants are complex values and Monad Algebra operators.

The thesis compares the higher-order languages based on the query languages Relational Algebra, Core XQuery, and Monad Algebra. Terms having no λ variables correspond to queries of the base query languages. We show the succinctness of our higher-order query languages, which can express queries that requires equivalent original queries of much bigger size. We also study several fundamental problems: evaluation, containment, and typing, for the higher-order query languages.

Since these higher-order query languages are more succinct than ordinary query languages, one should expect high complexity. The thesis investigates the complexity of the evaluation problem for the languages in detail. The complexity is shown for different classes of higher-order queries, which are classified by the order of variables that the queries contain. Even though the complexity of evaluating the queries is high, we show cases where the evaluation is not harder than evaluating Nonrecursive Datalog.

The second problem that the thesis has studied is the containment problem, which includes the containment between order 1 terms and the containment between order 2 terms. In the former case, terms have only relational variables that map to input values; whereas in the latter case, terms have both query variables and relational variables mapping to input values. When higher-order terms are of order 2, we consider containment between normalized terms to isolate the complexity due to λ variables. Our Π_2^{P}-complete complexity result (Theorem 6.8) implies that adding query variables does not make the containment problem harder.

The thesis also considers problems related to typing for higher-order terms. We show that adding higher-order variables and constants does not make the typeability problem more complicated. We also consider the typeability problem for weakly-typed HO terms, of which only the types of the constants are given.

The last contribution of the thesis is an implementation of a prototype that supports evaluation of higher-order terms. The system allows users to experiment with writing higher-order queries and comparing the performance over different techniques of reduction and materialization. Using higher-order variables and constants, users can write queries in a modular way and reuse both ordinary and higher-order queries. We have provided optimization techniques based on sharing of subterms, which decreases the complexity.

9.2 Future work

The containment problem between XQH terms.
We only study the containment problem for HO and HOCV terms because it is not clear what the correct definition of containment between XQH terms is. For HOCVP terms, we do not have tight bounds for containment of degree 1 order 1 terms. Getting tight bounds will help determine the exact complexity of the containment problem between two Monad Algebra expressions, which is still open.

Higher-order containment problem for CQ base.
In the thesis, we study the containment problem for higher-order terms over CQ base, which does not contain union, for some special cases, e.g., when the query variables are boolean, when there is no nesting of query variables, and when query variables are absent in one side. The general case of the containment problem is still open.

Higher-order containment problem between unnormalized terms.
When studying the higher-order containment problem, we only considered normalized

terms. For unnormalized terms, we can apply β-reduction first, then consider the containment between reduced terms. The upper bounds obtained by this method, however, are not tight. We plan to consider this problem carefully to obtain the exact complexity of the containment problem between unnormalized terms of order 2.

Typing - polymorphism.
The languages defined in the thesis do not allow higher-order variables that take inputs that range over many types. There are a number of works studying polymorphic type inference for relational data, e.g., [Mil78, BO96, VdBW02b]. This kind of problem for the complex object model and XML has received much interest recently, e.g., [VdBV07, HFC09]. Moreover, a primitive form of polymorphism can be implemented using "typeswitch", which is included in XQuery 3.0 [RCDSb]. A higher-order language that supports this also helps users to write queries in a more flexible way. For example, a higher-order function that returns attribute A of an input relation only requires that the input contains an attribute A, and does not have any requirement about the other attributes of the relation.

Well-definedness problem for higher-order queries.
Van den Bussche and his colleagues consider the "well-definedness problem", which is a more lenient and less syntactic condition than typeability by allowing some subterms to be untypeable, for languages without higher-order variables [Van05a, VdBVGV05, VdBVGV07b, VdBVGV07a]. In future work, we want to consider this problem for higher-order queries.

Evaluation techniques.
There are a number of possible techniques to improve the current implementation of the evaluation system.

1. Integrate with other Datalog optimization techniques, e.g., Magic Sets [CCIL09]: Higher-order queries containing only relational variables can be transformed to NRDL (see Proposition 4.1), but we have not used evaluation techniques for Datalog in the implementation. In the next version of the implementation, these techniques should be used to improve performance.

2. Integrate with other functional programming techniques: Since the current system uses a simple form of graph sharing for β-reduction, we want to employ into our implementation more complicated techniques of graph sharing, e.g., in [Gue05], which summarizes graph reduction techniques and the difficulty of

sharing implementations. Improving the interaction between β-reduction techniques and database evaluation techniques should improve the performance of the system.

3. Partial evaluation for XQH: In the current implementation, only subterms of order 0, which evaluate to XML documents, are materialized. A more general technique of materialization that can partially evaluate subterms of order 1 could be developed. Given an XQuery term, one can evaluate its subterms maximally and store the intermediate results. This will evaluate XQH terms more interactively by using the results of subterms as soon as the subterms are evaluated.

References

[AG98] A. Asperti and S. Guerrini. *The Optimal Implementation of Functional Programming Languages*. Cambridge University Press, 1998.

[AHV95] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[ASU79a] 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.

[ASU79b] A. V. Aho, Y. Sagiv, and J. D. Ullman. Equivalences among relational expressions. *SIAM Journal on Computing*, 8(2):218–246, 1979.

[Avi03] J. Avigad. Eliminating Definitions and Skolem Functions in First-order Logic. *ACM Transactions on Computational Logic*, 4(3):402–415, 2003.

[AYCS02] S. Amer-Yahia, S. Cho, and D. Srivastava. Tree Pattern Relaxation. In *EDBT*, 2002.

[Ban78] F. Bancilhon. On the completeness of query languages for relational data bases. In *MFCS*, 1978.

[BCF+] R. Busse, M. Carey, D. Florescu, M. Kersten, I. Manolescu, A. Schmidt, and F. Waas. XMark - An XML benchmark project. `http://www.xml-benchmark.org`.

[BCF+02] M. Benedikt, C. Y. Chan, W. Fan, R. Rastogi, S. Zheng, and A. Zhou. DTD-directed publishing with attribute translation grammars. In *PVLDB*, 2002.

[BCF03] V. Benzaken, G. Castagna, and A. Frisch. CDuce: an XML-centric general-purpose language. *ACM SIGPLAN Notices*, 38(9):51–63, 2003.

[BG10] M. Benedikt and G. Gottlob. The impact of views on containment. In *VLDB*, 2010.

[BK09] M. Benedikt and C. Koch. From XQuery to relational logics. *ACM Transactions on Database Systems*, 34(4):1–48, 2009.

[BMS11] H. Björklund, W. Martens, and T. Schwentick. Conjunctive query containment over trees. *Journal of Computer and System Sciences*, 77(3):450 – 472, 2011.

[BO96] P. Buneman and A. Ohori. Polymorphism and type inference in database programming. *ACM Transactions on Database Systems*, 21(1):30–76, 1996.

[BPV10] M. Benedikt, G. Puppis, and H. Vu. Positive higher-order queries. In *PODS*, 2010.

[BV12] M. Benedikt and H. Vu. Higher-order functions and structured datatypes. In *WebDB*, 2012.

[CCIL09] F. Calimeri, S. Cozza, G. Ianni, and N. Leone. Magic sets for the bottom-up evaluation of finitely recursive programs. In *LPNMR*, 2009.

[CDO09] B. Cautis, A. Deutsch, and N. Onose. Querying data sources that export infinite sets of views. In *ICDT*, 2009.

[CDOV11] B. Cautis, A. Deutsch, N. Onose, and V. Vassalos. Querying XML data sources that export very large sets of views. *ACM Transactions on Database Systems*, 36(1):5, 2011.

[CFP82] M. A. Casanova, R. Fagin, and C. H. Papadimitriou. Inclusion dependencies and their interaction with functional dependencies. In *PODS*, 1982.

[CFP84] 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.

[CKW93] 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.

[CLWY07] E. Cooper, S. Lindley, P. Wadler, and J. Yallop. Links: Web programming without tiers. In *FMCO*, 2007.

[CM77] A. K. Chandra and P. M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *STOC*, 1977.

[Cod72] E. F. Codd. *Relational completeness of data base sublanguages*. Database Systems, Prentice Hall and IBM Research Report RJ 987, 1972.

[Coo09] E. Cooper. The script-writers dream: How to write great SQL in your own language, and be sure it will succeed. In *DBPL*, 2009.

[Cou] Transaction Processing Performance Council. TPC-H Benchmark. `http://www.tpc.org`.

[DEGV01] E. Dantsin, T. Eiter, G. Gottlob, and A. Voronkov. Complexity and expressive power of logic programming. *ACM Computing Surveys*, 33(3):374–425, 2001.

[DG97] O. M. Duschka and M. R. Genesereth. Answering recursive queries using views. In *PODS*, 1997.

[DHT04] X. Dong, A. Y. Halevy, and I. Tatarinov. Containment of nested XML queries. In *VLDB*, 2004.

[FCB08] A. Frisch, G. Castagna, and V. Benzaken. Semantic subtyping: Dealing set-theoretically with function, union, intersection, and negation types. *Journal of the ACM*, 55(4):19:1–19:64, 2008.

[FGK07] W. Fan, F. Geerts, and X. J. A. Kementsietsidis. Rewriting regular XPath queries on XML views. In *ICDE*, 2007.

[FGPG09] 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.

[GHS07] C. Grün, A. Holupirek, and M. H. Scholl. Visually exploring and querying XML with BaseX. In *BTW*, 2007.

[GLS02] G. Gottlob, N. Leone, and F. Scarcello. Hypertree decompositions and tractable queries. *Journal of Computer and System Sciences*, 64(3):579–627, 2002.

[GM99] G. Grahne and A. O. Mendelzon. Tableau techniques for querying information sources through global schemas. In *ICDT*, 1999.

[GMRS09] T. Grust, M. Mayr, J. Rittinger, and T. Schreiber. FERRY: Database-supported program execution. In *SIGMOD*, 2009.

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

[GTL89] J.Y. Girard, P. Taylor, and Y. Lafont. *Proofs and types*. Cambridge tracts in theoretical computer science. Cambridge University Press, 1989.

[Gue05] S. Guerrini. Sharing implementations of graph rewriting systems. *Electronic Notes in Theoretical Computer Science*, 127(5):113–132, 2005.

[Hal00] A.Y. Halevy. Theory of answering queries using views. *ACM SIGMOD Record*, 29(4):40–47, 2000.

[HFC09] H. Hosoya, A. Frisch, and G. Castagna. Parametric polymorphism for xml. *ACM Transactions on Programming Languages and Systems*, 32(1):2:1–2:56, 2009.

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

[Hin08] J. R. Hindley. *Basic Simple Type Theory*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2008.

[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.

[HP03] H. Hosoya and B. C. Pierce. XDuce: a statically typed XML processing language. *ACM Transactions on Internet Technology*, 3(2):117–148, 2003.

[HPVD04] J. Hidders, J. Paredaens, R. Vercammen, and S. Demeyer. A light but formal introduction to XQuery. In *XSYM*, 2004.

[Imm86] N. Immerman. Relational queries computable in polynomial time. *Information and control*, 68(1-3):86–104, 1986.

[JK84] 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.

[Joh90] D. S. Johnson. *A catalog of complexity classes*. Handbook of theoretical computer science (vol. A). MIT Press, 1990.

[KLTV06] N. Koudas, C. Li, A. Tung, and R. Vernica. Relaxing join and selection queries. In *VLDB*, 2006.

[Klu88] A. Klug. On conjunctive queries containing inequalities. *Journal of the ACM*, 35(1):146–160, 1988.

[Koc06] 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.

[Koz77] D. Kozen. Lower Bounds for Natural Proof Systems. In *FOCS*, 1977.

[LMSS95] A. Levy, A. Mendelzon, Y. Sagiv, and D. Srivastava. Answering queries using views. In *PODS*, 1995.

[LRO96a] A. Y. Levy, A. Rajaraman, and J. J. Ordille. Query-answering algorithms for information agents. In *AAAI*, 1996.

[LRO96b] A. Y. Levy, A. Rajaraman, and J. J. Ordille. Querying heterogeneous information sources using source descriptions. In *VLDB*, 1996.

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

[LS97] A. Y. Levy and D. Suciu. Deciding containment for queries with complex objects (extended abstract). In *PODS*, 1997.

[LSS93] 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.

[Mai89] H. G. Mairson. Deciding ML typability is complete for deterministic exponential time. In *POPL*, 1989.

[Mai92] H. G. Mairson. A simple proof of a theorem of Statman. *Theoretical Computer Science*, 103(2):387–394, 1992.

[Mil78] R. Milner. A theory of type polymorphism in programming. *Journal of computer and system sciences*, 17(3):348–375, 1978.

[NSV07] A. Nash, L. Segoufin, and V. Vianu. Determinacy and rewriting of conjunctive queries using views: A progress report. In *ICDT*, 2007.

[NVdBVGV99] 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.

[OB88] A. Ohori and P. Buneman. Type inference in a database programming language. In *LFP*, 1988.

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

[ODPC06] N. Onose, A. Deutsch, Y. Papakonstantinou, and E. Curtmola. Rewriting nested XML queries using nested views. In *SIGMOD*, 2006.

[Par78] J. Paredaens. On the expressive power of the relational algebra. *Information Processing Letters*, 7(2):107–111, 1978.

[Pez99] E. Pezzoli. Computational complexity of Ehrenfeucht-Fraïsse games on finite structures. In *CSL*, 1999.

[PH01] R. Pottinger and A. Halevy. MiniCon: A scalable algorithm for answering queries using views. *The VLDB Journal*, 10:182–198, 2001.

[PVG92] J. Paredaens and D. Van Gucht. Converting nested algebra expressions into flat algebra expressions. *ACM Transactions on Database Systems*, 17(1):65–93, 1992.

[PW78] M. S. Paterson and M. N. Wegman. Linear unification. *Journal of Computer System Science*, 16(2):158–167, 1978.

[RCDSa] J. Robie, D. Chamberlin, J. Dyck, and J. Snelson. XQuery 1.1.: An XML query language. `http://www.w3.org/TR/2009/WD-xquery-11-20091215`.

[RCDSb] 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`.

[Ros92] K. A. Ross. Relations with relation names as arguments: Algebra and calculus. In *PODS*, 1992.

[Ros94] K.A. Ross. On negation in HiLog. *The Journal of Logic Programming*, 18(1):27–53, 1994.

[Sch01] A. Schubert. The complexity of β-reduction in low orders. In *TLCA*, 2001.

[Sne10] J. Snelson. Higher order functions for XQuery, 2010. Personal communication.

[Sta79] R. Statman. The typed λ-calculus is not elementary recursive. *Theoretical Computer Science*, 9(1):73–81, 1979.

[SV05] L. Segoufin and V. Vianu. Views and queries: Determinacy and rewriting. In *PODS*, 2005.

[SWK+02] A. Schmidt, F. Waas, M. Kersten, M. J. Carey, I. Manolescu, and R. Busse. XMark: A benchmark for XML data management. In *VLDB*, 2002.

[SY80] Y. Sagiv and M. Yannakakis. Equivalences among relational expressions with the union and difference operators. *Journal of the ACM*, 27(4):633–655, 1980.

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

[Ulr11] A. Ulrich. A FERRY-based query backend for the LINKS programming language. Master's thesis, University of Tübingen, 2011.

[Van05a] S. Vansummeren. Deciding well-definedness of XQuery fragments. In *PODS*, 2005.

[Van05b] S. Vansummeren. On the complexity of deciding typability in the Relational Algebra. *Acta Informatica*, 41(6):367–381, 2005.

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

[VB11a] H. Vu and M. Benedikt. Complexity of Higher-Order Queries. In *ICDT*, 2011.

[VB11b] H. Vu and M. Benedikt. HOMES: A higher-order mapping evaluation system. In *VLDB*, 2011.

[VdBV07] J. Van den Bussche and S. Vansummeren. Polymorphic type inference for the named nested relational calculus. *ACM Transactions on Computational Logic*, 9(1), 2007.

[VdBVGV05] 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*, 2005.

[VdBVGV07a] J. Van den Bussche, D. Van Gucht, and S. Vansummeren. A crash course on database queries. In *PODS*, 2007.

[VdBVGV07b] 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.

[VdBW02a] 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.

[VdBW02b] 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.

[VdM92] R. Van der Meyden. *The complexity of querying indefinite information: Defined relations, recursion and linear order*. PhD thesis, Rutgers University, 1992.

[VP00] V. Vassalos and Y. Papakonstantinou. Expressive Capabilities Description Languages and Query Rewriting Algorithms. *The Journal of Logic Programming*, 43(1):75 – 122, 2000.

[VV97] S. Vorobyov and A. Voronkov. Complexity of nonrecursive logic programs with complex values. Technical report, MPI-I-97-2-010, Max-Planck Institut fiir Informatik, Saarbriicken, 1997.

[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.

[Wah] D. Wahlin. XML data containing flight routes. `http://www.xmlforasp.net/codesection.aspx?csID=109`.

[Wan87a] M. Wand. Complete type inference for simple objects. In *LICS*, 1987.

[Wan87b] M. Wand. A simple algorithm and proof for type inference. *Fundamenta Informaticae*, 10(2):115–121, 1987.

[Won00] L. Wong. Kleisli, a functional query system. *Journal of Functional Programming*, 10(1):19–56, 2000.

[Woo93] P. T. Wood. Bottom-up evaluation of DataHiLog. In *RIDS*, 1993.

[Yan81] M. Yannakakis. Algorithms for acyclic database schemes. In *VLDB*, 1981.

Index

abstract types, 38

bucket algorithm, 154

conjunctive query containment, 19
    under dependencies, 19
construction tree
    HO, 32
    XQH, 147
containment problem
    HO, 84
    HOCV, 85
Core XQuery, 45, 46

degree
    HO, 27
    HOCV, 45
    XQH, 41

evaluation problem, 51

graph reduction, 138, 140

higher-order containment problem, 86
higher-order queries
    XQH, 37
higher-order query
    HO, 24
    HOCV, 42
    XQH, 37
HO term hierarchy, 29

linear HO term, 67

normal form, 28
NRDL, 21
NRDL$^\neg$, 21

order
    HO, 27
    HOCV, 45
    XQH, 38, 41

query language with inductive rules, 20

Relational Algebra, 19
rule-based conjunctive query, 18

TA$[t(n), a(n)]$, 22

un-nested term
    HO, 67
    XQH, 79

XML, 11