Higher-Order Functions and Structured Datatypes

Michael Benedikt
Department of Computer Science
Oxford University
Parks Road, Oxford, UK
michael.benedikt@cs.ox.ac.uk

Huy Vu
Department of Computer Science
Oxford University
Parks Road, Oxford, UK
huy.vu@cs.ox.ac.uk

ABSTRACT

Recent proposals from the World Wide Web consortium propose adding support for higher-order functions within the XQuery standard. In this work we explore languages adding higher-order features on top of XML and other structured datatypes. We define a higher-order extension for Core XQuery, along with a higher-order algebra over complex values which has the same complexity as the XML-based language. We discuss our language and its relation with proposed extensions to the XQuery standard, study the complexity of evaluation, and briefly discuss our approach to implementing the language.

1. INTRODUCTION

Higher-order functions play a fundamental role in computer science, and most functional programming languages feature them. While relational query languages, such as SQL, have not generally had a close connection to functional programming languages, the main XML query language, XQuery is functional, descending from earlier work on functional query languages. As such, it is natural to consider support for higher-order queries – transformations of queries, transformation of transformations of queries, etc. – within XQuery. And indeed, the proposed next iteration of the standard, XQuery 3.0 [9], supports higher-order functions. Moreover, higher-order functions have been included in a number of XQuery processors including Saxon-PE 9.3 and BaseX 7.1.1 [1].

The motivating scenario for higher-order functions in XQuery is a traditional one: to gain modularity. The following example is taken from a whitepaper by an XQuery working group member on adding higher-order query support to XQuery [10]. 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.

[1]http://basex.org/

EXAMPLE 1. *We define the generic function.*

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("books.xml")/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 [9] 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 data integration and XML access control [6]. 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 2. *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* $\$seq$. *The reduced form of* $query_2$ *is as follows.*

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

Although there is no formal semantics for XQuery 3.0, in several implementations we have looked at, such as BaseX, 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 (albeit with less modularity) 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)
};
```

In this paper, we will present a higher-order XML query language, denoted $\mathsf{XQ_H}$, extending the Core XQuery language of Koch [8]. We will also look at higher order query languages for other "structured datatypes", defining a higher-order language on complex values, denoted HOCV, and compare it to $\mathsf{XQ_H}$.

There is already a known connection between Core XQuery, Complex-valued languages, and λ-calculi. Koch [8] 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. But Monad Algebra (as Core XQuery) does not allow abstraction over queries – it has higher-order constants but no higher-order variables. In our previous work [3, 14], we considered the combination of database queries and a more extensive λ-calculus, allowing abstraction over queries, queries over queries, etc. However, our prior work did not support complex values or XML. Here we will thus look to bridge the more general higher-order querying framework of [3, 14] with the support for complex values and XML. We will focus on the evaluation problem for both of these higher-order languages for "structured data". We show that support for query variables comes "almost for free" in these languages. Support for arbitrary order variables leads to non-elementary complexity, but with a natural restriction again we get no increase in the worst-case bounds.

Organization. Section 2 defines the higher-order variant of XQuery, $\mathsf{XQ_H}$. Section 3 gives HOCV, a higher-order complex-valued language, and discusses the relationship with $\mathsf{XQ_H}$. Section 4 shows the complexity of evaluating HOCV terms containing variables of arbitrary order. In Section 5 we give conclusions, briefly discuss our prototype implementation, and related work. Due to space limitations, proofs are omitted, but can be found in [13].

Acknowledgement Benedikt is supported in part by EPSRC EP/H017690/1 (the Engineering and Physical Sciences research council, UK) and by the European Commission FET-Open project FOX, FP7-ICT-233599.

2. A HIGHER-ORDER EXTENSION OF CORE XQUERY

Before presenting our language, we define the set of *abstract types*, and their *order* which plays a large rule in the syntax.

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

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

Each abstract type has a *denotation*; the base type maps to sequences of nodes within unranked labeled ordered trees. Additional XML features, such as attributes, can be coded into this extension. This coding does not impact evaluation. 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 abstract type we assume a set of variables associated with it. We are now ready to give the syntax of the higher-order extension of Core XQuery:

$$\begin{array}{rcl} query & := & () \mid \langle a \rangle query \langle /a \rangle \mid query\ query \\ & \mid & var \mid var/axis :: \nu \\ & \mid & \text{for } var \text{ in } query \text{ return } query \\ & \mid & \text{if } cond \text{ then } query \\ & \mid & [\mathcal{V}]query \\ & \mid & query(query) \\ \mathcal{V} & := & \emptyset \mid var, \mathcal{V} \\ cond & := & query = query \mid query \end{array}$$

In the syntax above, 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 $[\doteq]$), which compares labels of two leaves, or (b) deep equality (denoted $[\cong]$), an isomorphism test of two nodes (identified with their subtrees).

The syntax of $\mathsf{XQ_H}$ is based roughly on that of Koch's Core XQuery (reviewed in the next section), 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 the 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 [8].

Before giving the semantics of $\mathsf{XQ_H}$, we describe a function AT assigning abstract types to each $\mathsf{XQ_H}$ query. We give the inductive definition for several cases; the rules for other cases can be easily inferred from the syntax of the queries. 1. AT of each query without an abstraction is $\mathcal{B}$. 2. AT of a query with a nonempty abstraction [] is defined as: $\mathsf{AT}([x, \mathcal{V}]query) = \mathsf{AT}(x) \to \mathsf{AT}([\mathcal{V}]query)$ 3. If $\mathsf{AT}(query_1) = \mathcal{T}_1 \to \mathcal{T}_2$ and $\mathsf{AT}(query_2) = \mathcal{T}_1$, then $\mathsf{AT}(query_1 \ @\ query_2) = \mathcal{T}_2$. 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". In this work, we will always assume that queries are well-typed. Note that, as shown in Example 2, XQuery 3.0 adopts a finer type system, where (e.g.) one can specify that a variable binds to a sequence as opposed to a single node.

The semantics of XQ_H 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. We omit the rules for equality above: these state that an equality returns a fixed non-empty tree exactly when the two terms satisfy the corresponding equality (atomic or deep).

In addition, to the rules above, we can transform terms via standard β-reduction: e.g. $([x]q_1)@q'$ can be transformed to $q_1[x \mapsto q']$ in a subterm. Although this 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 3. *Let $\$D$ and $\$R$ be two variables that have* AT *equal to $\mathcal{B}$, $\$Q$ be a query variable that has* AT *equal to $\mathcal{B} \rightarrow \mathcal{B}$. The query:*

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

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. In our complexity results, we will be interested in queries of order 0 – i.e. ones that evaluate to a document. XQ_H^m denotes the set of XQ_H queries containing variables of order at most m.

3. HIGHER-ORDER COMPLEX-VALUED QUERIES

We now define a corresponding language over complex values, basing it on the complex-valued query language Monad Algebra [11, 8]. 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. As with Higher-order XML, we start with the type system. We fix an infinite linearly-ordered set of *attribute names* (or *attributes*). We associate with each attribute name A_i a range $\mathcal{D}om(A_i)$ 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.

Although we do not consider boolean attributes here, we do have 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 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 $\mathcal{T} \rightarrow \mathcal{T}'$ is a type with denotation the set of functions from $\mathcal{D}$ to $\mathcal{D}'$, whose order is

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

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

- We will include constants for all nested relational instances, referred to as *nested relational constants*.
- We consider the following order 1 constants – i.e. *query constants*;
  - singleton set construction of type $\mathcal{T} \to \{\mathcal{T}\}$;
  - flatten of type $\{\{\mathcal{T}\}\} \to \{\mathcal{T}\}$;
  - pairing of type $\langle A_1 : \{\mathcal{T}_1\}, \ldots, A_n : \mathcal{T}_n\rangle \to \{\langle A_1 : \mathcal{T}_1, \ldots, A_n : \mathcal{T}_n\rangle\}$;
  - for each nested relational type containing a type A_i the unary projection operator π_{A_i} of type $\langle A_1 : \mathcal{T}_1, \ldots, A_n : \mathcal{T}_n\rangle \to \mathcal{T}_i$;
  - for any type $\mathcal{T}$ the binary operator $\cup$, which returns the union of two order 0 terms of type $\mathcal{T}$;
  - for each relational type $\mathcal{T}$ the unary selection operator $\sigma_{A_i=A_j}$, which selects a subset of the tuples from a given nested relation of type $\{A_1 : \tau_1, \ldots, A_n : \tau_n\}$, where $=$ is either (a) "atomic equality" [$\doteq$], that is, a label comparison, or (b) [$\cong$] isomorphism on nested relations;
  - for each set of attribute names $A_1 \ldots A_n$ tuple formation with these attributes
- Lastly, we consider an order 2 constant, named *map*, of type $(\mathcal{T} \to \mathcal{T}') \to \{\mathcal{T}\} \to \{\mathcal{T}'\}$

Terms. Higher-order *terms* are built up from the constants above and variables 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 semantics of terms uses a set operational rules, including the standard λ-calculus rules for β-reduction and application, plus rules for constants at order 0:

$$\frac{X \Rightarrow \{\tau_1, \ldots, \tau_n\}}{flatten(X) \to \tau_1 \cup \ldots \cup \tau_n}$$

$$\frac{X \Rightarrow \langle A_1 : \tau_1, \ldots, A_n : \tau_n\rangle}{pairwith_{A_1}(X) \to \{\langle A_1 : \rho_1, \ldots, A_n : \tau_n\rangle \mid \rho_1 \in \tau_1\}}$$

$$\frac{\tau \Rightarrow \langle A_1 : \tau_1, \ldots, A_n : \tau_n\rangle}{\pi_{A_i}(X) \to \tau_i}$$

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

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

We omit the rules for selection, union, and singleton above.

As with XQ_H there are conditions for a term to be well-typed. We omit a full discussion of this here, and from now on assume that terms are well-typed. In giving the syntax above we have implicitly considered typed versions of every operator, and assumed explicit types for every variable; in this case checking well-typedness is fairly straightforward. If we do not assume this, then we have a typing problem for the language which is more complicated than the XML version – e.g. if we project on attribute a, then we must be sure the term we project on must have an a attribute in it. Still it can be shown that standard algorithms for typing in the simply typed λ-calculus (e.g. Wand's algorithm [16]) can be extended to this languages: see [13] for details.

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.

We denote HOCV^k (resp. $\mathsf{HOCV}^k_\mathsf{L}$) the fragment of HOCV (resp. HOCV_L) where the order of variables is bounded by k.

We also define the *size* of a term. 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.

The Complex-Value/XML correspondence at higher-order. 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:

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

From the syntax, we can see that XQ is a special case of XQ_H where higher-order variables are absent. On the complex-valued side, Koch considered Monad Algebra on lists in [8]. This is equivalent to $\mathsf{HOCV}^0_\mathsf{L}$, 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 XQ^0_H and $\mathsf{HOCV}^0_\mathsf{L}$.

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

PROPOSITION 1. *Given $k \geq 0$, evaluating XQ^k_H queries and evaluating $\mathsf{HOCV}^k_\mathsf{L}$ terms are polynomially reducible.*

The proposition is shown by giving a translation between a HOCV^k term and an XQ^k_H query, which is an extension of the original translation in [8]. The details of the translation are given in [13].

The bottom line is that from now on we can study the complexity of HOCV_L, and derive the complexity of XQ_H from these results. The advantage is that HOCV_L is easier to analyze, since the syntax is simpler and more compositional. 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.

4. COMPLEXITY OF THE EVALUATION PROBLEM

The evaluation problem considered in this section is defined as follows. The *boolean query evaluation problem* takes as input a well-typed HOCV term of boolean query type, along with a set of nested relational constants. The output is true iff the application of the term on the instances evaluates to the (unique) nonempty instance of $\{\langle\rangle\}$.

We will be interested in the *combined complexity* of the problem, where the size of the input is the database size plus the term size, as well as the *query complexity*, in which the database instance is fixed. When the *query* is fixed, higher-order features can be compiled away, and the complexity is thus the same as the data complexity of the corresponding base language (e.g. Core XQuery): in all cases, these are known from Koch's work [8] to be in polynomial time.

We first review the complexity of evaluation for HOCV^0 terms, i.e. terms with abstraction over nested relational variables. We say "review", because the equivalence result in the previous section and Koch's complexity results in [8] give bounds on their evaluation:

PROPOSITION 2. *The evaluation problem for* HOCV^0 *with* $[\doteq]$ *without negation (resp. with negation) is* NEXPTIME*-complete (resp.* $\mathsf{TA}[2^{\mathsf{O}(\mathsf{n})}, \mathsf{O}(\mathsf{n})]$*-complete).* $\mathsf{TA}[2^{\mathsf{O}(\mathsf{n})}, \mathsf{O}(\mathsf{n})]$ *refers to he class of alternating machines that uses (linearly) exponential amount of time but only linearly many alternations – see [8].*

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

Note that there is still a gap in the complexity of evaluating HOCV^0 with $[\cong]$, which is $\mathsf{TA}[2^{\mathsf{O}(\mathsf{n})}, \mathsf{O}(\mathsf{n})]$-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 3. *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.*

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 4. *The evaluation problem of* HOCV^1 *with either* $[\doteq]$ *or* $[\cong]$ *is* EXPSPACE*-complete.*

For the upper bound, we show that an evaluation strategy based on full β-reduction followed by a standard evaluation procedure for Core XQuery will use exponential space. Proposition 3 gives a bound on the size of tree representations emerging in normalization, and the bound now follows from a corresponding bound on Core XQuery evaluation for terms of this size. The lower bound uses a method for coding machine computations in XML documents from Koch's [8] – however it makes use of an iteration technique that is specific to the higher-order setting.

For terms containing variables of order 2, the complexity goes up significantly, to 2-EXPTIME. In general, we can ascertain the complexity of evaluating HOCV terms of arbitrary degree, using techniques from [14].

THEOREM 5. *Given* $k \geq 1$*, the problem of evaluating* HOCV^k *with either* $[\doteq]$ *or* $[\cong]$ *is:*

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

The membership is shown by first reducing variables of higher order using β-reduction, then recursively evaluating the term containing variables of lower order. The hardness is shown analogously to the proof of Proposition 13 in [14], which shows similar bounds in the relational case, but roughly one exponential lower than the ones here. This uses the "Church numeral" technique well-known in the λ-calculus literature [1]: using higher-order queries to implement (hyper)-exponential iteration. The main difference from the relational bounds stems from the fact 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.

HOCV	XQuery	Negation	Eq.	Complexity
HOCV^0	XQ	no	$[\doteq]$	NEXPTIME
		yes	$[\doteq]$	$\mathsf{TA}[2^{\mathsf{O}(\mathsf{n})}, \mathsf{O}(\mathsf{n})]$
		no/yes	$[\cong]$	**in** EXPSPACE
HOCV^1	XQ^1_H	no/yes	$=$	EXPSPACE
HOCV^2	XQ^2_H	no/yes	$=$	2-EXPTIME
HOCV^{2m-1}	$\mathsf{XQ}^{2m-1}_\mathsf{H}$	no/yes	$=$	m-EXPSPACE
HOCV^{2m}	$\mathsf{XQ}^{2m}_\mathsf{H}$	no/yes	$=$	(m+1)-EXPTIME

Table 1: Complexity of Evaluation

Table 1, with Eq. an abbreviation of Equality, summarizes the complexity of fragments of XQuery, Monad Algebra and their higher-order analogs. 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. All the complexity results in the table are complete except for the case of HOCV^0 and XQ with deep equality, which have a $\mathsf{TA}[2^{\mathsf{O}(\mathsf{n})}, \mathsf{O}(\mathsf{n})]$ lower bound and EXPSPACE upper bound. When the language contains variables of order 1, the complexity of the evaluation problem becomes EXPSPACE-complete.

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 our prior work on the relational case [14].

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.

A term is self-nested free (or un-nested) if it does not contain any self-nested variable. Intuitively, a term is un-nested if a variable never occurs as an argument of itself in the term. 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 6. *The evaluation problem for un-nested* HOCV^n *terms with* $n \geq 1$ *is* EXPSPACE*-complete.*

The algorithm underlying this result is again basically β-reduction. The key idea (also present in [14]) is that during reduction, the size of a term may increase but the height of the term changes only linearly – this in turn implies an exponential bound on the size of every intermediate term produced. This last fact allows an algorithm that guesses and verifies each intermediate result in EXPSPACE.

Theorem 6 follows from a more general result, that allows the nesting restriction to hold only above some order.

THEOREM 7. *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* $(m+1)/2$-EXPSPACE*-complete if* m *is odd or* $(m/2+1)$-EXPTIME*-complete if* m *is even, which are the bounds for* HOCV^m.

5. DISCUSSION

We have considered an extension of Core XQuery and a complex-valued query language to include higher-order transformations. Even though the evaluation problem is non-elementary in the general case, we showed that restrictions on nesting lead to drastic reductions in the complexity of the evaluation problem.

Theory and Practice. The previous results show that the complexity of query evaluation is caused by the ability of nested higher-order queries to enable "massive sharing of subexpressions". We admit that the worst case examples used in the proofs (i.e. "Church numerals") are extremely artificial. Still the ability to use higher-order features to succinctly express complex queries is in principle a feature, not a bug, and we would like to allow users to exploit it as much as possible.

Currently we have a prototype implementation of XQ_H that works on top of the XML query engine BaseX. The engine works by alternating β-reductions and ordinary XQuery evaluation. The main feature present in the engine to deal with high complexity is *graph reduction*, the same one used in our prior work in the relational case [15]. Intermediate queries are stored as DAGs, rather than trees; while performing β-reduction, rather than always copy a subterm that is repeated multiple times, we look for opportunities to share it. At the end of reduction, the final DAG is evaluated bottom-up with the granularity of XML queries used in the bottom-up evaluation determined via a cost-based refinement algorithm, reminiscent of the one used in [2]. Details can be found in [13]. The main new difficulty here is in the cost-based analysis. While for relational higher-order queries, [15] relies on cost estimates provided by an underlying relational engine, for XML query engines robust cost estimation interfaces are not readily available. Thus we currently rely on manually-provided cost information.

There are several limitations of our language that are worth highlighting. First of all, XQuery 3.0 adds many interesting features in addition to higher-order functions, such as more powerful switch expressions, grouping operators, and exception handling. The interaction of these with higher-order features still needs to be examined. Although we motivated our work by support for greater modularity and incremental development of XQuery transformations, the language presented here is monolithic. In our implementation we are exploring interactive development of XML transformations, along with partial evaluation.

Related Work. Our work is extremely closely related to Koch's work concerning Core XQuery and its fragments and their relationship to complex-valued languages [8] . Koch gave bounds on the complexity of evaluating those fragments, along with reductions between evaluation problems for Core XQuery and Monad Algebra. Our XQ_H and HOCV are extensions of XQ and Monad Algebra, respectively, that support higher-order queries. Our complexity bounds can be seen as extension of those in [8] – indeed, we combine the techniques from our prior work on relational higher-order queries [14] with those of Koch for XQuery.

Recently, Cooper [4] 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 [12] has described an implementation that uses the FERRY framework [7] to translate a subset of the LINKS programming language [5], 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. Unlike in our work, no complexity bounds are given in works like [4]. But unlike these works, we have not implemented the higher-order nested relational language, only the XML and relational versions; our implementation focuses on one optimization technique, while [4, 5] deal with a wide range of implementation issues.

6. REFERENCES

[1] A. Beckmann. Exact Bounds for Lengths of Reductions in Typed λ-Calculus. *J. Symb. Log.*, 66(3):1277–1285, 2001.

[2] M. Benedikt, C. Y. Chan, W. Fan, R. Rastogi, S. Zheng, and A. Zhou. DTD-directed publishing with attribute translation grammars. In *PVLDB*, 2002.

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

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

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

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

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

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

[9] J. Robie, D. Chamberlin, M. Dyck, and J. Snelson. XQuery 3.0: An XML Query Language. *W3C Working Draft*, 2010.

[10] J. Snelson. Higher order functions for XQuery, 2010. personal communication.

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

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

[13] H. Vu. *Higher-Order Queries and Applications*. PhD thesis, Oxford University, 2012. Available at `http://www.cs.ox.ac.uk/people/huy.vu/PhDThesis.pdf`.

[14] H. Vu and M. Benedikt. Complexity of higher-order queries. In *ICDT*, 2011.

[15] H. Vu and M. Benedikt. HOMES: A higher-order mapping evaluation system. *PVLDB*, 4(12), 2011.

[16] M. Wand. A simple algorithm and proof for type inference. *Fundamenta Infomaticae*, 10:115–122, 1987.