Deciding Well-Definedness of XQuery Fragments

Stijn Vansummeren*
Limburgs Universitair Centrum
Universitaire Campus - Gebouw D
B-3590 Diepenbeek, Belgium
stijn.vansummeren@luc.ac.be

ABSTRACT

Unlike in traditional query languages, expressions in XQuery can have an undefined meaning (i.e., these expressions produce a run-time error). It is hence natural to ask whether we can solve the well-definedness problem for XQuery: given an expression and an input type, check whether the semantics of the expression is defined for all inputs adhering to the input type. In this paper we investigate the well-definedness problem for non-recursive fragments of XQuery under a bounded-depth type system. We identify properties of base operations which can make the problem undecidable and give conditions which are sufficient to ensure decidability.

1. INTRODUCTION

Much attention has been paid recently to XQuery, the XML query language currently under development by the World Wide Web Consortium [6, 11]. Unlike in traditional query languages, expressions in XQuery can have an undefined meaning (i.e., these expressions produce a run-time error). As an example, consider the following variation on one of the XQuery use cases [9]:

```
<bib> {
  for $b in $bib/book
  where $b/publisher = "ACM"
  return element{$b/author}{$b/title}
} </bib>
```

This expression should create, for each book published by ACM, a node whose name equals the author of the book and whose child is the title of the book. If there is a book with more than one `author` node however, then the result of this expression is undefined because the XQuery specification requires that the first argument to the element constructor is a singleton sequence.

This leads us to the natural question whether we can solve the *well-definedness problem* for XQuery: given an expression and an input type, check whether the semantics of the expression is defined for all inputs adhering to the input type. This problem is undecidable for any computationally complete programming language, and hence also for XQuery. Following good programming language practice, XQuery is therefore equipped with a static type system (based on XML Schema [5, 20]) which ensures "type safety" in the sense that every expression which passes the type system's tests is guaranteed to be well-defined. Due to the undecidability of the well-definedness problem, such type systems are necessarily incomplete. That is, there are expressions which are well-defined, but not well-typed.

The typical XQuery expression does not use the whole of XQuery's computational power however. For example, sixty-two out of the seventy-seven XQuery uses cases [9] can be written as a "for-let-where-return" expression without recursive function definitions. It is hence worthwhile to investigate whether we can solve the well-definedness problem for such expressions. If so, then we essentially obtain a type-system which is both sound and complete. We note that the XQuery type system remains incomplete on this restricted fragment. Indeed, consider the following expression which is trivially well-defined since the then-branch of the if-test will never be executed.

```
if false then element{()}{()} else ()
```

In the general setting for which the XQuery type system is designed, it is undecidable to check that an expression always evaluates to true. The XQuery type system is therefore "conservative" in the sense that it requires both branches of an if-test to type-check. Since the then-branch in our example is always undefined, it cannot type-check, and hence the whole expression is ill-typed. This example clearly illustrates that in order to solve well-definedness one also has to solve "satisfiability". We will see, however, that this alone is not sufficient to solve well-definedness.

In previous work [21] we investigated the well-definedness problem for *Relational XQuery* (RX). This is a first-order, set-based database dialect of XQuery without recursive functions; where we only allowed the child axis; took a value-based point of view (i.e., we ignored node identity); and

*Research Assistant of the Fund for Scientific Research - Flanders (Belgium)

PODS 2005 June 13-15, 2005, Baltimore, Maryland.

observed a type system similar to that of the nested relational or complex object data model [1, 8, 22]. Since RX can simulate the relational algebra, well-definedness is still undecidable, even in this restricted setting. Surprisingly, the problem remains undecidable for the positive-existential fragment of RX. This undecidability is due to the identification of an item with the singleton containing that item in the XQuery data model [13], which becomes difficult to analyze under a set-based semantics. Removing this identification leads to a dialect of positive-existential RX for which well-definedness is decidable.

The above results do not transfer to XQuery itself however. Indeed, the XQuery data model is sequence-based instead of set-based, has node-identity, and its type system is far richer than that of the complex object data model. In this paper we therefore study the well-definedness problem in the XQuery data model. Specifically, we study a family of query languages XQ(B) under a bounded-depth version of the XQuery type system. Here, B is a set of *base operations* (such as for example XQuery's atomic value comparison, its children axis, ...) and XQ(B) is the query language obtained from B by adding variables, constants, conditional tests, let-bindings, and for-loops. As such, every XQ(B) is a fragment of "for-let-where-return" XQuery without recursive function definitions. We also omit type-tests (such as XQuery's *instanceof* and *typeswitch*) since these quickly make the well-definedness problem undecidable [21]. The bounded-depth restriction of the XQuery type system is motivated by the fact that most XML documents in practice have nesting depth at most five or six, and that unbounded-depth nesting is hence often not needed.

Concretely, we identify properties of base operations which can make the well-definedness problem undecidable and give corresponding conditions which are sufficient to ensure decidability. The decidability of well-definedness for a large fragment of XQuery immediately follows as we show that, in the absence of automatic coercions, the various axis movements, node constructors, value and node comparisons, and node-name and text-content inspections satisfy these conditions. In contrast, well-definedness for this fragment with automatic coercions is undecidable.

Related work

Our formalization of XQuery is very similar to the independently developed one by Hidders et al. [14].

As we will see that the presence of atomic data values significantly influences the well-definedness problem, we restrict ourselves in what follows to the related work which also considers data values.

A problem reminiscent to the well-definedness problem is *semantic type-checking*: check that the output of a given expression is always in a given output type for every input adhering to a given input type [2, 3]? We will show however, that there are XQ(B) for which well-definedness is decidable, but semantic type-checking is not.

We will also show that in order to solve well-definedness it is necessary (but not sufficient) to solve the satisfiability problem (i.e., non-empty output of a well-defined expression on at least one input). Recently, Lakshmanan et al. [17] have studied the computational complexity of satisfiability in tree pattern queries, which capture a fragment of XPath (and hence of XQuery).

Organization

This paper is further organized as follows. We introduce our abstraction of the XQuery data model in Section 2. In Section 3 we introduce the notion of a base base operation and show how to extend these to a query language XQ(B). In Section 4 we state the well-definedness problem, introduce a bounded-depth version of the XQuery type system, identify several properties of base operations which may render the well-definedness problem undecidable, and propose corresponding restrictions on base operations. We show that these restrictions are sufficient to ensure decidability in Section 5 and conclude in Section 6.

2. DATA MODEL

XQuery expressions do not operate directly on XML text, but on instances of the XQuery data model [13]. Every value in this data model is a sequence of items, where every item is an atomic value or a node. There are seven node kinds, the most prominent being the element, attribute, and text nodes. Nodes are grouped in "hedges" (sequences of trees). In what follows we will restrict ourselves to values containing only atoms, element nodes, and text nodes. This restriction is done solely for simplicity, we could add the remaining node types without sacrificing any of our results.

The XQuery type system is an integral part of the XQuery data model as every item in a value also carries a *type annotation*. Examples of such annotations are `integer` (for atoms) and `element of type Bibliography` (for element nodes). Potentially, these type annotations can also be `untypedAtomic` (for atoms) or `untyped` (for nodes) indicating that the item was not validated against a schema. We will restrict ourselves in what follows to such unvalidated values. This decision is motivated as follows. XQuery uses the type annotations of validated inputs during (1) static and dynamic type-checking[1] and (2) the evaluation of type-tests (such as *instance-of* and *typeswitch*). The aim of this paper is to study well-definedness, which is more fundamental than static or dynamic type-checking. Furthermore, we will omit type-tests in our study, as these quickly turn the well-definedness problem undecidable [21]. It is therefore safe to restrict ourselves to unvalidated values. All references to the semantics of XQuery should hence be understood to mean "the semantics of XQuery when the input is unvalidated". As atoms can now only carry the type annotation `untypedAtomic` and nodes can only carry the annotation `untyped`, we can drop type annotations altogether in our formalization.

Atoms and nodes

Formally, we assume to be given a recursively enumerable set $\mathcal{A} = \{a, b, \dots\}$ of *atoms*, which contains the booleans *true* and *false*. We further assume to be given an infinite set $\mathcal{N} = \{n, m, \dots\}$ of *nodes*, disjoint with $\mathcal{A}$, which is partitioned

[1]Static type-checking is an optional feature in XQuery. All XQuery processors have to perform dynamic type-checking however.

into a recursively enumerable, infinite set $\mathcal{N}^e$ of *element nodes* and a recursively enumerable, infinite set $\mathcal{N}^t$ of *text nodes*. Elements of $\mathcal{A} \cup \mathcal{N}$ are called *items*.

Stores

Nodes are given an interpretation inside an *XML store* [14, 16]. This is essentially a sequence of ordered node-labeled trees, where each tree represents an XML fragment. For example, Figure 1(a) depicts a store where the first tree represents the XML fragment in Figure 1(b) and the second tree represents the XML fragment in Figure 1(c). Here we use circles to depict element nodes and boxes to depict text nodes. Formally, a store Σ is a tuple $(V, E, \lambda, <, \prec)$ where

- V is a finite set of nodes;
- E is the *edge relation*: a binary relation on V such that (V, E) is an acyclic directed graph where every node has in-degree at most one and text nodes have out-degree zero (hence (V, E) is composed of trees);
- $\lambda : V \to \mathcal{A}$ is the *labeling function* which associates each node in V with its *label*;[2]
- $<$ is the *sibling order*: a strict partial order on V that compares exactly the different children of a common node:

  $$(n < n') \vee (n' < n) \Leftrightarrow \exists m \in V : E(m, n) \wedge E(m, n');$$

  and
- $\prec$ is a strict partial order on the roots, i.e., the nodes with in-degree zero.

Using the sibling and root order, the XQuery data model defines the *document order* $\ll$ on Σ which intuitively equals the left-to-right, pre-order traversal of a sequence of trees. Formally, the document order $\ll$ is the strict total order on V such that (1) if $E(n, n')$ then $n \ll n'$, and (2) if $m < n$ or $m \prec n$, $E^*(m, m')$, and $E^*(n, n')$, then $m' \ll n'$. Here we write E^* for the reflexive transitive closure of E.

Example 1. Figure 1(a) depicts the store $(V, E, \lambda, <, \prec)$ where

$$\begin{aligned} V &= \{n_1, n_2, n_3, n_4, n_5, n_6, n_7, n_8, n_9\} \\ E &= \{(n_1, n_2), (n_1, n_4), (n_2, n_3), (n_5, n_6), (n_5, n_8), \\ &\qquad (n_6, n_7), (n_8, n_9)\} \\ < &= \{(n_2, n_4), (n_6, n_8)\} \\ \prec &= \{(n_1, n_5)\}, \end{aligned}$$

and where λ is defined by

$$\begin{array}{lll} \lambda(n_1) := \text{beer} & \lambda(n_2) := \text{name} & \lambda(n_3) := \text{Duvel} \\ \lambda(n_4) := \text{blond} & \lambda(n_5) := \text{name} & \lambda(n_6) := \text{first} \\ \lambda(n_7) := \text{John} & \lambda(n_8) := \text{last} & \lambda(n_9) := \text{Doe}. \end{array}$$

For this store $n_i \ll n_j$ if, and only if, i is less than j. □

[2]The label of a text node is called the node's *content* in XQuery terminology.

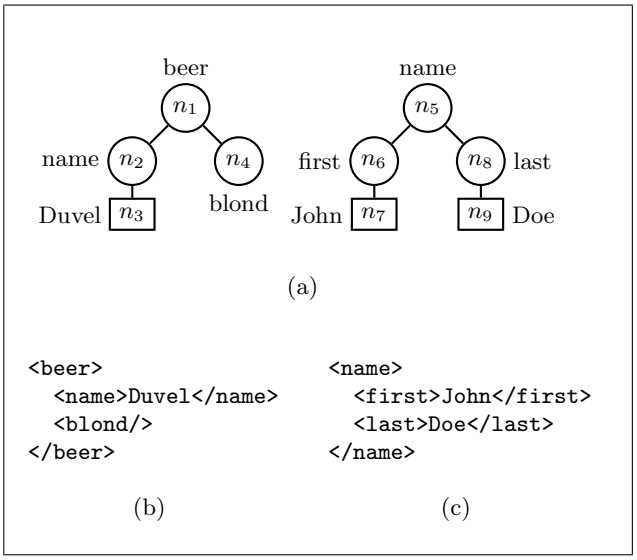


Figure 1: A store and the XML fragments it represents.

We will use the standard terminology for trees on stores. That is, if $E(m, n)$ then m is the *parent* of n and n is a *child* of m. A node $n \in V$ is a *root node* of Σ if it has in-degree zero. We write $\mathit{roots}(\Sigma)$ for the set of all root nodes in Σ. If Σ has at most one root node, then we say that Σ is a *tree*. Note that the empty store is hence also a tree. For convenience we will denote the empty store by $\varnothing$.

Two stores Σ and Σ' are *disjoint* when $V_\Sigma \cap V_{\Sigma'} = \emptyset$. If Σ and Σ' are disjoint stores then the *concatenation of* Σ and Σ', denoted by $\Sigma \circ \Sigma'$, is the store with node set $V_\Sigma \cup V_{\Sigma'}$, edges $E_\Sigma \cup E_{\Sigma'}$, labeling function $\lambda_\Sigma \cup \lambda_{\Sigma'}$, sibling order $<_\Sigma \cup <_{\Sigma'}$, and root order $\prec_\Sigma \cup \prec_{\Sigma'} \cup\ \mathit{roots}(\Sigma) \times \mathit{roots}(\Sigma')$. Clearly, all stores can be written as a concatenation of trees.

Finally, if n is a node in Σ, then the *sub-tree of* Σ *rooted at* n, denoted by $\Sigma|_n$, is the store with nodes

$$V' = \{m \mid E^*(n, m)\},$$

edges $E \cap (V' \times V')$, labeling function $\lambda|_{V'}$, sibling order $< \cap (V' \times V')$, and the empty root order.

Values

A *value-tuple* of arity p is a tuple $(\Sigma; s_1, \ldots, s_p)$ where Σ is a store and every s_j is a finite sequence of atoms and nodes in Σ. A *value* is a value-tuple of arity one. We write $\mathcal{V}_p$ for the set of all value-tuples with arity p and abbreviate $\mathcal{V}_1$ by $\mathcal{V}$.

We denote the empty sequence by $\langle\rangle$, non-empty sequences by for example $\langle a, b, c\rangle$, and the concatenation of two sequences s_1 and s_2 by $s_1 \circ s_2$. In addition, we will write $s(j)$ for the j-th item of a sequence s, $|s|$ for the length of s, $\mathit{rng}(s)$ for the set of items occurring in s, and $\vec{s}$ for a tuple $s_1, \ldots, s_p$ of sequences.

Renamings

A *renaming* ρ is a permutation of $\mathcal{A} \cup \mathcal{N}$ that is the identity on the booleans and maps atoms to atoms, element

nodes to element nodes, and text nodes to text nodes. A *node-renaming* is a renaming that is the identity on atoms. Renamings are extended to sets, tuples, and sequences in the canonical way:

$$\rho(S) = \{\rho(v) \mid v \in S\}$$
$$\rho((v_1, \ldots, v_p)) = (\rho(v_1), \ldots, \rho(v_p))$$
$$\rho(\langle v_1, \ldots, v_p\rangle) = \langle \rho(v_1), \ldots, \rho(v_p)\rangle$$

Note that in particular ρ is thus also extended to stores and value-tuples.

Two value-tuples v and v' are *isomorphic*, denoted by $v \equiv v'$, when there exists a renaming ρ such that $\rho(v) = v'$. Two value-tuples v an v' are *node-isomorphic*, denoted by $v \equiv_{node} v'$, when there exists a node-renaming such that $\rho(v) = v'$.

3. SYNTAX AND SEMANTICS

We are interested in studying the well-definedness problem for fragments of "for-where-let-return" XQuery. The (un)decidability of the well-definedness problem for a certain fragment depends of course on the set of allowed operations in this fragment. In order to establish fundamental properties of these operations which cause (un)decidability, we introduce the formal concept of a *base operation*. Base operations serve as a general framework in which the concrete behavior of XQuery's basic operations and functions [18] can be described.

Base operations

Formally, a *base operation* of arity p is a relation $R \subseteq \mathcal{V}_p \times \mathcal{V}$ which is

1. *Computable*: it is effectively decidable, given a value-tuple v, whether there exists a w such that $R(v, w)$, and if so, such a w is effectively computable from v.
2. *Store-increasing*: R only relates value-tuples $(\Sigma; \vec{s})$ to values of the form $(\Sigma \circ \Sigma'; s')$ with Σ' possibly empty. Hence, R can add trees to a store, but cannot modify existing trees.
3. *Node-generic*: for every node-renaming ρ we have
$$R(v, w) \Leftrightarrow R(\rho(v), \rho(w)).$$
As such, R can only interpret nodes by the information given in the input store. Furthermore, nodes which are added to the input store are chosen non-deterministically.
4. a *Semi-function*: R is a function up to node-isomorphism: if $R(v, w)$ and $R(v, z)$ then $w \equiv_{node} z$.
5. *Reachable-only*: R only uses information of those trees in the input store whose nodes are mentioned in one of the input sequences. That is, for all sequence-tuples $\vec{s}$, all (possibly empty) trees $\Theta_1, \ldots, \Theta_k, \Theta'_1, \ldots, \Theta'_k$ such that $\Theta_j = \Theta'_j$ if a node of Θ_j is mentioned in $\vec{s}$, and all stores Σ disjoint with $\Theta_1, \ldots, \Theta_k, \Theta'_1, \ldots, \Theta'_k$, we have
$$R((\Theta_1 \circ \cdots \circ \Theta_k; \vec{s}), (\Theta_1 \circ \cdots \circ \Theta_k \circ \Sigma; s')) \Leftrightarrow R((\Theta'_1 \circ \cdots \circ \Theta'_k; \vec{s}), (\Theta'_1 \circ \cdots \circ \Theta'_k \circ \Sigma; s'))$$

We write $R(v)$ for the set of all values w for which $R(v, w)$ holds. The first four properties above capture the notion of a "determinate" transformation from the theory of object-creating queries [1]. As such, $R(v)$ is finitely representable and this representation can effectively be computed from v.

Let us give some examples of base operations:

- XQuery's concatenation operator *concat* (written as a comma in XQuery) is a binary base operation that relates $(\Sigma; s, s')$ to $(\Sigma; s \circ s')$.
- XQuery's *children* axis is a unary base operation that relates $(\Sigma; s)$ with s a sequence of nodes to $(\Sigma; s')$ where s' is the unique sequence containing the children of nodes in s in document order. Formally,
$$rng(s') = \{n \mid \exists m \in rng(s) : E(m, n)\},$$
and $s'(i) \ll s'(j)$ when $i < j$. Note that there are no repeated nodes in s', since $\ll$ is a strict order. XQuery's other axes (i.e., *parent*, *descendant*, *following-sibling*, ...) can similarly be viewed as unary base operations.
- XQuery's atomization function *data* can be modelled as a unary base operation that relates $(\Sigma; s)$ to $(\Sigma; s')$ where s' has the same length as s, and $s'(j)$ is the coercion of $s(j)$ to an atom. That is, $s'(j) = s(j)$ when $s(j)$ is an atom, $s'(j) = \lambda(s(j))$ if $s(j)$ is a text node, and $s'(j) = \mathit{fold}(r)$ if $s(j)$ is an element node. Here, r is the unique sequence containing all text node descendants of $s(j)$ in document order and *fold* is an abstract function mapping sequences of text nodes to atoms. In XQuery, *fold* returns the string concatenation of the text nodes' labels.
- XQuery's atomic value comparison *eq* is a binary base operation that relates $(\Sigma; s, s')$ to $(\Sigma; \langle\rangle)$ if s or s' is the empty sequence, and relates $(\Sigma; \langle a\rangle, \langle b\rangle)$ to $(\Sigma; \langle a = b\rangle)$.[3] We will use a C-style notation for comparisons: $a = b$ evaluates to *true* when a equals b, and evaluates to *false* otherwise.
- XQuery's node comparisons *is* and $\ll$ are binary base operations that relate $(\Sigma; s, s')$ to $(\Sigma; \langle\rangle)$ if s or s' is the empty sequence, and relate $(\Sigma; \langle n\rangle, \langle m\rangle)$ to $(\Sigma; \langle n = m\rangle)$ respectively $(\Sigma; \langle n \ll m\rangle)$.
- XQuery's kind tests *is-element* and *is-text* are unary base operations that relate $(\Sigma; \langle n\rangle)$ to $(\Sigma; \langle n \in \mathcal{N}^e\rangle)$ respectively $(\Sigma; \langle n \in \mathcal{N}^t\rangle)$.[4]
- XQuery's *node-name* function is a unary base operation that relates $(\Sigma; \langle\rangle)$ to $(\Sigma; \langle\rangle)$, relates $(\Sigma; \langle n\rangle)$ to $(\Sigma; \langle\lambda(n)\rangle)$ when $n \in \mathcal{N}^e$, and relates $(\Sigma; \langle n\rangle)$ to $(\Sigma; \langle\rangle)$ when $n \in \mathcal{N}^t$. Conversely, we could also consider an operation *content* which behaves like *node-name*, but then on text nodes.

[3] XQuery's atomic value comparison will actually first atomize its arguments using the *data* function described earlier, and then compare the obtained sequences according to our semantics. This behavior can be simulated in our query language XQ(B) defined below as it allows composition of base operations.

[4] Kind tests are part of XPath expressions in XQuery, and are written as for example `$x/self::element()` or `$x/self::text()`.

- XQuery's element node constructor *element* is a binary base operation that relates $(\Sigma; \langle a \rangle, s)$ to $(\Sigma \circ \Theta, \langle n \rangle)$ where Θ is a tree, disjoint with Σ, whose root element node n is labeled by a, such that there exists a bijection g from $[1, |s|]$ to the children of n satisfying[5]
  1. $g(j) <_\Theta g(j')$ if j is less than j';
  2. $g(j)$ is a text node labeled with $s(j)$ if $s(j)$ is an atom; and
  3. $\Theta|_{g(j)} \equiv_{node} \Sigma|_{s(j)}$ if $s(j)$ is a node.
- XQuery's text node constructor *text* is a unary base operation that relates $(\Sigma; \langle a \rangle)$ to $(\Sigma \circ \Theta, \langle n \rangle)$ where Θ is a tree, disjoint with Σ, whose root text node n is labeled by a.
- A final example of a unary base operation is XQuery's emptiness test function *empty* which relates $(\Sigma; s)$ to $(\Sigma; \langle \mathit{true} \rangle)$ when $s = \langle\rangle$, and relates $(\Sigma; s)$ to $(\Sigma; \langle \mathit{false} \rangle)$ otherwise.

Expressions

We create a query language XQ(B) out of a finite set of base operations B by adding variables, constants, and basic control-flow as follows. For each base operation R we assume to be given a *base expression* f: a unique syntactical entity which denotes R. For ease of notation we will often not distinguish between a base operation and its associated base expression. As such, we will write for example $v \in f(w)$ to denote $v \in R(w)$.

The syntax of XQ(B) is defined by the following grammar:

$$\begin{array}{rcl} e & ::= & x \mid a \mid () \mid f(e_1, \ldots, e_p) \\ & \mid & \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \\ & \mid & \texttt{let } x := e_1 \texttt{ return } e_2 \\ & \mid & \texttt{for } x \texttt{ in } e_1 \texttt{ return } e_2 \end{array}$$

Here, e ranges over expressions, x ranges over variables, a ranges over atoms, f ranges over base expressions in B, and p is the arity of f. We view expressions as abstract syntax trees and omit parentheses. The set $FV(e)$ of *free variables* of an expression e is defined as usual. That is, the free variables of x is $\{x\}$, the free variables of a and $()$ is the empty set, the free variables of $f(e_1, \ldots, e_p)$ and `if` e_1 `then` e_2 `else` e_3 is the union of the free variables of their immediate subexpressions, and the free variables of `let` $x := e_1$ `return` e_2 and `for` x `in` e_1 `return` e_2 is

$$FV(e_1) \cup (FV(e_2) \setminus \{x\}).$$

Semantics

The input to an expression e is described by a *context* $(\Sigma; \sigma)$ on e, consisting of a store Σ and a function σ from a finite superset $\{x, \ldots, y\}$ of the free variables of e to sequences of items such that $(\Sigma; \sigma(x), \ldots, \sigma(y))$ is a value-tuple. A function from a finite set of variables to sequences of items is called an *environment*. We write $x : s, \sigma$ for the environment σ' with domain $dom(\sigma) \cup \{x\}$ such that $\sigma'(x) = s$ and $\sigma'(y) = \sigma(y)$ for $y \neq x$.

[5]The semantics of XQuery's element constructor is actually more complex in the sense that adjacent text nodes are grouped into one node by concatenating their content. Since we do not wish to give a concrete interpretation to the atoms, we discard this behavior.

The semantics of an expression e is described by means of the *evaluation relation*, as defined in Figure 2. Here, we write $(\Sigma; \sigma) \models e \Rightarrow (\Sigma'; s)$ to denote the fact that e evaluates to value $(\Sigma'; s)$ on context $(\Sigma; \sigma)$ on e. We note that the disjointness requirements in the rules for base operation invocation and for loop ensure that different invocations of a subexpression add different nodes to the input store. We will write $e(\Sigma; \sigma)$ for the set of all values to which e can evaluate on context $(\Sigma; \sigma)$. It is easy to see that the semantics of an expression only depends on its free variables: if two environments σ and σ' are equal on $FV(e)$, then

$$(\Sigma; \sigma) \models e \Rightarrow (\Sigma'; s) \quad \Leftrightarrow \quad (\Sigma; \sigma') \models e \Rightarrow (\Sigma'; s).$$

Example 2. XPath expressions like `$bib/book` can be simulated in XQ(*children*, *is-element*, *node-name*, *eq*) as follows

```
for $b in children($bib) return
  if is-element($b) then
    if eq(node-name($b), "book") then $b else ()
  else ()
```

□

Example 3. XQuery's quantified expressions can be simulated using the emptiness test. For example,

```
some $x in data($pubs) satisfies $x eq "ACM"
```

can be expressed in XQ(*eq*, *data*, *empty*) as follows:

```
let $z :=
  for $x in data($pubs) return
    if eq($x,"ACM") then $x else ()
return
  if empty($z) then false else true
```

It immediately follows that generalized comparisons (such as `$pubs = "ACM"`) can also be simulated using the emptiness test, as such comparisons are just syntactic sugar for quantified expressions like the one shown above. □

Example 4. If B contains the base operations *children*, *data*, *eq*, *node-name*, and *element*, then the XQuery expression from the beginning of Section 1 can be expressed in XQ(B) as follows.

```
element("bib",
  for $b in $bib/book return
    if $b/publisher = "ACM" then
      element(data($b/author), $b/title)
    else ()
)
```

For the sake of brevity we have not expanded XPath expressions (such as `$bib/book`) or generalized comparisons (such as `$b/publisher = "ACM"`), as we have already shown how to simulate these in the previous examples. □

When we fix some order on the variables, a context $(\Sigma; \sigma)$ with $dom(\sigma) = \{x, \ldots, y\}$ is fully determined by the value-tuple $(\Sigma; \sigma(x), \ldots, \sigma(y))$. Since the semantics of an expression only depends on its free variables, every expression e

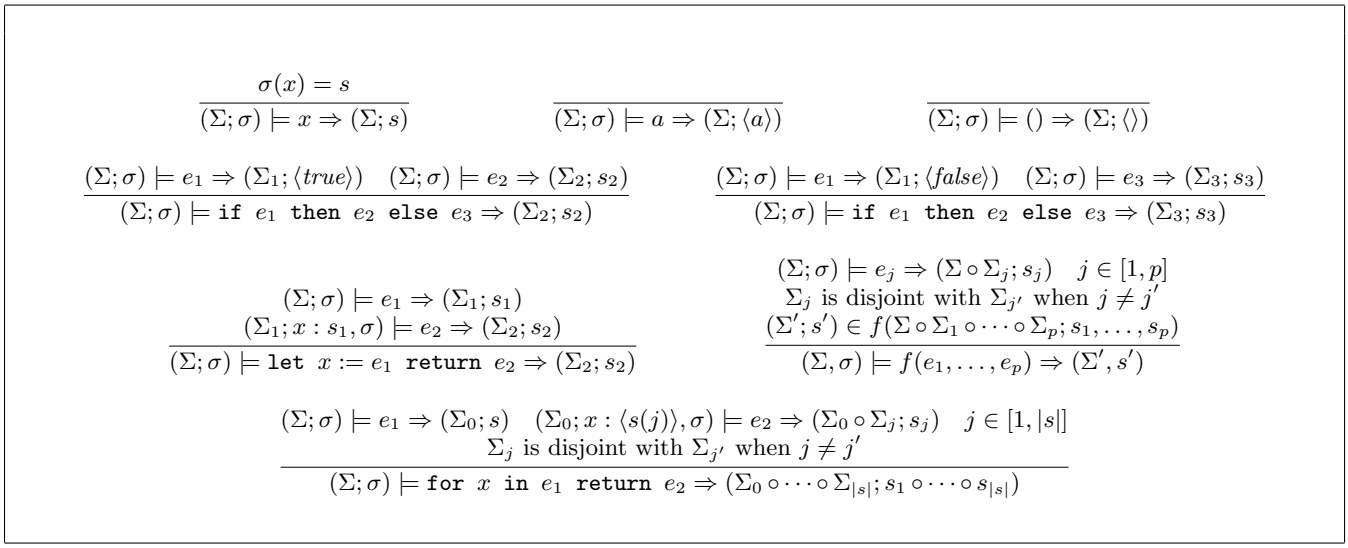


Figure 2: The evaluation relation

hence defines a relation on $\mathcal{V}_p \times \mathcal{V}$, where p is the number of free variables in e. It is then possible to show that every expression defines a computable, store-increasing, node-generic, reachable-only semi-function.

Proposition 5. *Every expression e in XQ(B) defines a base operation.*

4. WELL-DEFINEDNESS

The evaluation of an expression e on an input $(\Sigma;\sigma)$ may be *undefined*, i.e., $e(\Sigma;\sigma) = \emptyset$. For instance, the expression of Example 4 returns the empty result when `$bib` contains a book node which is published by ACM and is written by multiple authors. Indeed, the element constructor does not relate any value to value-tuples of the form $(\Sigma; s, s')$ with s not a singleton atom.

Since the fact that e is undefined on $(\Sigma;\sigma)$ models the situation where an actual implementation would produce a run-time error, it is a natural question to ask wether we can check, given an expression e and a (possibly infinite) set S of contexts on e, whether e is defined on every context in S. The answer to this problem clearly depends on both the set B of base operations and the class of context sets used as inputs. For the purpose of this paper we will focus on the class of context sets specified by a bounded-depth version of the XQuery type system. This class is rich enough to model the typical data processing scenario of XQuery where the input consists of a tuple of input trees, which are processed starting from the top. Indeed, it is a public secret that most XML documents in practice have nesting depth at most five or six, and that unbounded-depth nesting is hence often not needed.

Formally, a *type* is a term generated by the following grammar:

$$\begin{array}{rcl} \tau & ::= & \mathbf{atom} \mid \mathbf{text} \mid \mathbf{element}(a,\tau) \\ & \mid & \mathbf{empty} \mid \tau + \tau \mid \tau \circ \tau \mid \tau^* \end{array}$$

A type denotes a set of values, as defined in Figure 3. Here we denote the empty store by $\varnothing$ and trees by Θ. For ease of notation we will not distinguish between a type and its denotation. A *type assignment* Γ on an expression e is a function from the free variables $\{x, \ldots, y\}$ of e to types. A type assignment denotes the set of contexts

$$\{(\Sigma_x \circ \cdots \circ \Sigma_y; \sigma) \mid (\Sigma_z; \sigma(z)) \in \Gamma(z) \text{ for all } z \in \{x, \ldots, y\}\}.$$

Again we will not distinguish between a type assignment and its denotation.

Definition 6. Let B be a finite set of base operations. We say that $e \in \mathrm{XQ}(B)$ is *well-defined* under a type-assignment Γ on e if $e(\Sigma;\sigma) \neq \emptyset$ for every context $(\Sigma;\sigma) \in \Gamma$. The *well-definedness problem for XQ(B)* consists of checking, for a given expression $e \in \mathrm{XQ}(B)$ and a given type assignment Γ on e, whether e is well-defined on Γ.

It is not obvious that the well-definedness problem is decidable. Indeed, we will next identify several properties of B which can make the problem undecidable.

4.1 Non-monotonic behavior

Definition 7. Let B be a finite set of base operations. Let e be an expression in $\mathrm{XQ}(B)$ and let Γ be a type assignment under which e is well-defined. We say that e *is satisfiable under* Γ if there exists a context $(\Sigma;\sigma) \in \Gamma$ such that s is non-empty for every value $(\Sigma'; s) \in e(\Sigma;\sigma)$. The *satisfiability problem* for $\mathrm{XQ}(B)$ consists of checking, given e and Γ, whether e is satisfiable under Γ.

Note that, since every expression defines a semi-function by Proposition 5, s is non-empty for some $(\Sigma'; s) \in e(\Sigma;\sigma)$ if, and only if, *all* values in $e(\Sigma;\sigma)$ have a non-empty list. Hence, e is satisfiable under Γ if, and only if, there exists a context $(\Sigma;\sigma) \in \Gamma$ and a value $(\Sigma'; s)$ in $e(\Sigma;\sigma)$ such that s is non-empty.

The satisfiability problem is reducible to the well-definedness problem. Indeed, let e be an expression in $\mathrm{XQ}(B)$ and let Γ

$$\frac{a \in \mathcal{A}}{(\varnothing; \langle a \rangle) \in \textbf{atom}} \qquad \frac{n \in \mathcal{N}^t \text{ is } \Theta\text{'s root}}{(\Theta, \langle n \rangle) \in \textbf{text}}$$

$$\frac{\begin{array}{c} n \in \mathcal{N}^e \text{ is } \Theta\text{'s root} \quad \lambda_\Theta(n) = a \\ E_\Theta(n) = \{n_1, \ldots, n_k\} \quad n_1 <_\Theta \ldots <_\Theta n_k \\ (\Theta|_{n_1} \circ \cdots \circ \Theta|_{n_k}; \langle n_1, \ldots, n_k \rangle) \in \tau \end{array}}{(\Theta, \langle n \rangle) \in \textbf{element}(a, \tau)} \qquad \frac{}{(\varnothing, \langle\rangle) \in \textbf{empty}}$$

$$\frac{(\Sigma, s) \in \tau_1 \text{ or } (\Sigma, s) \in \tau_2}{(\Sigma, s) \in \tau_1 + \tau_2} \qquad \frac{\begin{array}{c} (\Sigma_1, s_1) \in \tau_1 \quad (\Sigma_2, s_2) \in \tau_2 \\ \Sigma_1 \text{ is disjoint with } \Sigma_2 \end{array}}{(\Sigma_1 \circ \Sigma_2, s_1 \circ s_2) \in \tau_1 \circ \tau_2}$$

$$\frac{\begin{array}{c} (\Sigma_1, s_1) \in \tau \quad \cdots \quad (\Sigma_p, s_p) \in \tau \quad p \geq 0 \\ \Sigma_j \text{ is disjoint with } \Sigma_{j'} \text{ when } j \neq j' \end{array}}{(\Sigma_1 \circ \cdots \circ \Sigma_p, s_1 \circ \ldots \circ s_p) \in \tau^*}$$

Figure 3: The denotation of types.

be a type assignment under which e is well-defined. Then e is unsatisfiable under Γ if, and only if, the expression

`for` x `in` e `return` (`if` () `then` () `else` ())

is well-defined under Γ. Indeed, since the subexpression `if` () `then` () `else` () is always undefined, it follows that the whole expression is defined if, and only if, e always returns the empty list. In earlier work [21] we have already obtained the following proposition for a set-based data model, which parallels earlier results on semistructured query languages such as StruQL [12]:

Proposition 8. *If B includes the base operations concat, children, eq, node-name, content, element, and empty, then XQ(B) can simulate the relational algebra. Concretely, for every relational algebra expression ϕ over database schema $\mathbf{S}$ there exists $e_\phi \in XQ(B)$ and a type assignment $\Gamma_\mathbf{S}$ such that*

- *e_ϕ is well-defined under $\Gamma_\mathbf{S}$,*
- *$\Gamma_\mathbf{S}$ contains encodings of all databases over $\mathbf{S}$, and,*
- *e_ϕ evaluated on an encoding of database D equals an encoding of $\phi(D)$.*

Consequently, satisfiability (and hence well-definedness) is undecidable for XQ(B) above because it is undecidable for the relational algebra. Note, however, that every expression in the *monotone* fragment of the relational algebra (i.e., the relational algebra without difference where selection only tests equality on attributes) is satisfiable. The satisfiability problem for the monotone relational algebra is hence trivially decidable. In the hope of finding XQ(B) for which the well-definedness is decidable, it is therefore worthwhile to restrict ourselves to those base operations which are in a sense also "monotone". For this purpose, we adapt the notion of (unordered) complex object containment [4] to (ordered) values.

Containment

Intuitively, a store Σ is contained in a store Σ' if Σ can be obtained by removing nodes from Σ' in such a way that if we remove a node, we also remove all of its descendants. Formally, Σ is contained in Σ' when every component of Σ is a subset of the corresponding component of Σ' i.e., $V_\Sigma \subseteq V_{\Sigma'}$, $E_\Sigma \subseteq E_{\Sigma'}$, $\lambda_\Sigma \subseteq \lambda_{\Sigma'}$, $<_\Sigma \subseteq <_{\Sigma'}$, $\prec_\Sigma \subseteq \prec_{\Sigma'}$, and $roots(\Sigma) \subseteq roots(\Sigma')$.[6]

[6]Containment is closely related to the notion of *simulation* [7]: there exists a simulation from Σ to Π which respects document order and relates all roots of Σ to roots of Π if, and only if, there exists a store Σ', node-isomorphic to Π, such that Σ is contained in Σ'.

Example 9. Consider the three stores depicted in Figure 4. It is easy to verify that Σ_2 is contained in Σ_3. Store Σ_1 is not contained in Σ_2 however, as n_1 is a root in Σ_1, but not in Σ_2. It can similarly be seen that Σ_1 is also not contained in Σ_3. □

By the same intuition as above, a sequence s is contained in a sequence s' if s can be obtained from s' by deleting items. Formally, s is contained in s' if there exists a strictly increasing function $h : [1, |s|] \to [1, |s'|]$ such that $s(j) = s'(h(j))$ for every $j \in [1, |s|]$. Such a function h is called a *witness* of the fact that s is contained in s'.

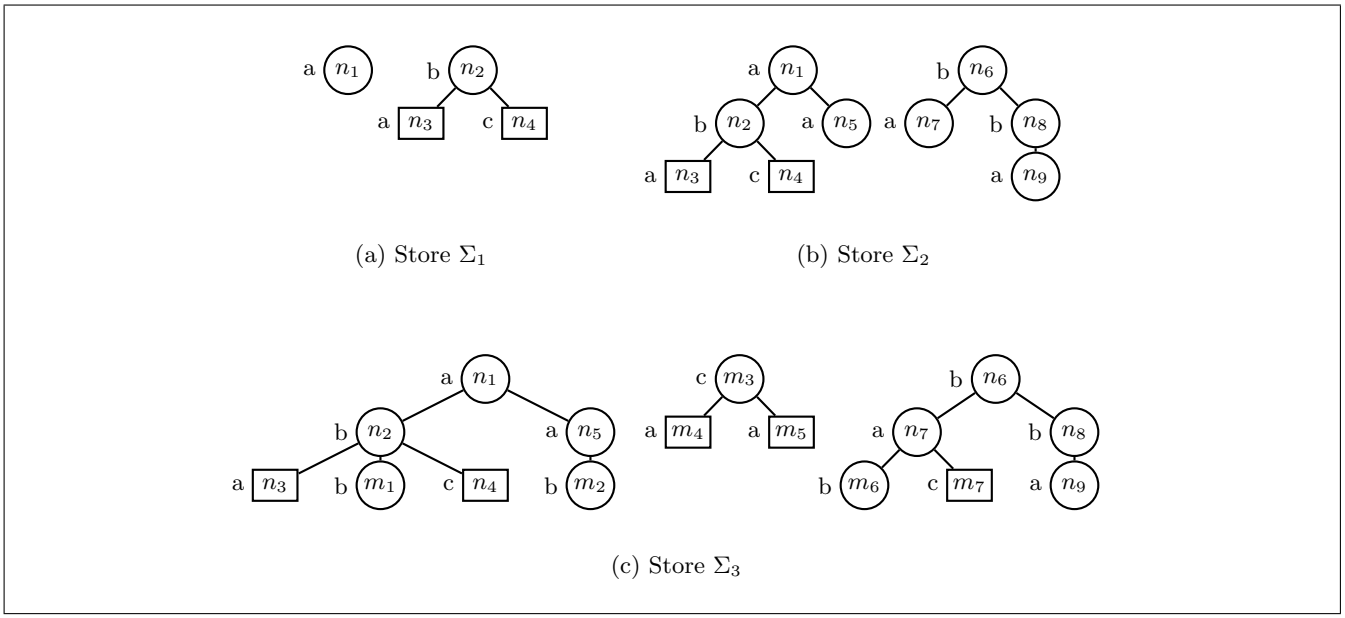
Example 10. Consider the sequences $s = \langle a, b, c \rangle$ and $s' = \langle a, b, b, a, c \rangle$. Then s is contained s', as witnessed by the function $h : [1, 3] \to [1, 5]$ with $h(1) = 1$, $h(2) = 2$, and $h(3) = 5$. In contrast, when $s = \langle a, a, b, c \rangle$ then s is not contained in s', as we cannot obtain s from s' simply by deleting items in s'. □

Containment extends naturally to value-tuples: a value-tuple $(\Sigma; s_1, \ldots, s_p)$ is contained in a second value-tuple $(\Sigma'; s'_1, \ldots, s'_p)$ if Σ is contained in Σ' and every s_j is contained in the corresponding s'_j. Finally, we lift the containment relation to sets of value-tuples: $S \sqsubseteq S'$ if, and only if, for every $v' \in S'$ there exists $v \in S$ such that v is contained in v'. In what follows we will denote the containment relation on stores, sequences, value-tuples and sets of value-tuples by $\sqsubseteq$.

Monotonicity

A relation $R \subseteq \mathcal{V}_p \times \mathcal{V}$ is *monotone* if for all v and w in $\mathcal{V}_p$ with $R(v) \neq \emptyset$, $R(w) \neq \emptyset$, and $v \sqsubseteq w$ we have $R(v) \sqsubseteq R(w)$.

Example 11. The concatenation operator *concat* is clearly monotone. The children axis is also monotone. Indeed, let $(\Sigma; s) \sqsubseteq (\Sigma'; s')$ and suppose that *children* is defined on $(\Sigma; s)$ and $(\Sigma'; s')$. Since $s \sqsubseteq s'$ we know that every node mentioned in s is also mentioned in s'. Furthermore, since $\Sigma \sqsubseteq \Sigma'$ we know the set of children of a node n in Σ is a subset of the set of children of n in Σ'. Finally, since $\Sigma \sqsubseteq \Sigma'$ we know that if n precedes m in document order in Σ, it also precedes m in document order in Σ'. Hence, if $(\Sigma; t)$ is the result of



(a) Store Σ_1

(b) Store Σ_2

(c) Store Σ_3

Figure 4: Containment of stores. Store Σ_1 is not contained in Σ_2 or in Σ_3. Store Σ_2 is contained in Σ_3.

children on (Σ, s) and $(\Sigma'; t')$ is the result of *children* on $(\Sigma'; s')$, then it is easily seen that $t \sqsubseteq t'$. Therefore,

$$children(\Sigma; s) = \{(\Sigma; t)\} \sqsubseteq \{(\Sigma'; t')\} = children(\Sigma'; s'),$$

as desired. □

We can generalize example 11 as follows:

Proposition 12. *The base operations concat, children, parent, preceding-sibling, following-sibling, descendant, ancestor, eq, is, is-element, is-text, $\ll$, node-name, content, element, and text are all monotone.*

Note that if our restriction to monotone base operations is to have any chance of leading to $XQ(B)$ for which well-definedness is decidable, *empty* must be non-monotone. Indeed, all other base operations mentioned in Proposition 8 are monotone by Proposition 12. Fortunately:

Example 13. The emptiness test is not monotone. Indeed, let Σ be a store. The emptiness test relates $(\Sigma; \langle\rangle)$ only to $(\Sigma; \langle true \rangle)$ and $(\Sigma; \langle a \rangle)$ only to $(\Sigma; \langle false \rangle)$. However, $\langle true \rangle \not\sqsubseteq \langle false \rangle$, although $(\Sigma; \langle\rangle) \sqsubseteq (\Sigma; \langle a \rangle)$. □

We also note that the atomization function *data*, depending on the concrete interpretation of the function *fold*, is potentially not monotone. Indeed, suppose for example that *fold*, when viewed as a base operation, relates $(\Sigma; \langle\rangle)$ to $(\Sigma; \langle a \rangle)$ and relates $(\Sigma; s)$ with s a non-empty sequence of text nodes to $(\Sigma; \langle b \rangle)$ for some $b \neq a$.[7] Then *fold* (and hence *data*) is clearly not monotone. Our definition of monotonicity is not being overly restrictive, since with this interpretation of *fold* we can simulate an emptiness test in XQ(B). Indeed, *empty*(e) is simulated by

$$eq(data(element(c, \texttt{for } x \texttt{ in } e \texttt{ return } c)), a).$$

[7]The actual interpretation of *fold* in XQuery has this kind of behavior: it relates the empty sequence to the empty string, and non-empty sequences to non-empty strings.

Here, c is an atom. As such, we obtain that XQ(*concat*, *children*, *eq*, *node-name*, *content*, *element*, *data*) is at least as expressive as XQ(*concat*, *children*, *eq*, *node-name*, *content*, *element*, *empty*). Its well-definedness problem is hence also undecidable. This reasoning clearly illustrates that automatic coercions, such as the one performed by atomization, are not harmless with regard to deciding well-definedness.

We note that monotonicity transfers from base operations to expressions:

Proposition 14. *If B is a finite set of monotone base operations then every expression in $XQ(B)$ defines a monotone relation.*

4.2 Interpretation of atoms

Another potential source of undecidability is the interpretation of atoms by base operations. Indeed, suppose that B includes base operations $+$ and $\times$ which interpret the atoms as integers and simulate the addition respectively multiplication on them. That is, $+$ and $\times$ relate $(\Sigma; \langle k \rangle, \langle l \rangle)$ to $(\Sigma; \langle k + l \rangle)$ respectively $(\Sigma; \langle k \times l \rangle)$. Note that $+$ and $\times$ are monotone. It is easy to see that for every polynomial $P(x_1, \ldots, x_k)$ with integer coefficients there exists an expression e_P with free variables $x_1, \ldots, x_k$ that simulates P. Hence, the expression

$$\texttt{if } eq(e_P, 0) \texttt{ then } (\texttt{if } () \texttt{ then } () \texttt{ else } ()) \texttt{ else } ()$$

is well-defined under the type assignment which maps every x_j to **atom** if, and only if, the Diophantine equation $P(x_1, \ldots, x_k) = 0$ has no integer solution. Since we now have a reduction from Hilbert's undecidable tenth problem [19], well-definedness for XQ(B) is undecidable.

We will therefore restrict ourselves to base operations which cannot interpret the atoms, except for the booleans *true* and *false*. Formally, we require base operations R to be *generic*: for every renaming ρ we have $w \in R(v) \Leftrightarrow \rho(w) \in R(\rho(v))$.

It is easy to see that all base operations mentioned in Section 3, except *data* (depending on the concrete interpretation of *fold*), are generic.

Proposition 15. *The base operations concat, children, parent, preceding-sibling, following-sibling, descendant, ancestor, eq, is, is-element, is-text, $\ll$, node-name, content, element, text, and empty are all generic.*[8]

Note that hence genericity alone is not powerful enough to prevent the construction of XQ(*concat*, *children*, *eq*, *node-name*, *content*, *element*, *empty*) for which well-definedness is undecidable.

An easy induction shows:

Proposition 16. *If B is a finite set of generic base operations, then for every expression $e \in XQ(B)$ and every renaming ρ which is the identity on the atoms mentioned in e we have $v \in e(\Sigma;\sigma) \Leftrightarrow \rho(v) \in e(\rho(\Sigma;\sigma))$.*

4.3 Non-local behavior

In this section we will show that, even if B is a set of monotone and generic base operations, well-definedness for XQ(B) need not be decidable. In order to illustrate this statement, we first note that the containment problem for XQ(*concat*) is undecidable:[9]

Proposition 17. *It is undecidable to check, given expressions e_1 and e_2 in XQ(concat) with the same set of free variables and a type assignment Γ under which e_1 and e_2 are well-defined, whether $e_1(v) \sqsubseteq e_2(v)$ for every $v \in \Gamma$. This is true, even when we only allow type assignments in which every context has depth at most one.*

The proof largely resembles that of Ioannidis and Ramakrishnan, who have shown that containment of unions of conjunctive queries on bags is undecidable [15].

The undefinedness behavior of base operations such as *eq*, *children*, and *element* is quite simple: one of the input sequences contains two or more items; the input sequence contains an atom where it should only contain nodes; or the first input sequence is not a singleton atom, respectively. Base operations with more complex undefinedness behavior are problematic with regard to well-definedness checking, as the following corollary to Proposition 17 shows.

Corollary 18. *Let check be the base operation which relates $(\Sigma; s, s')$ to $(\Sigma; \langle\rangle)$ if $s \sqsubseteq s'$ and which is undefined otherwise. The well-definedness problem for XQ(check,concat) is undecidable.*[10]

Crux. It is easy to see that expressions in XQ(*concat*) do not create new nodes. Therefore, if e is an expression in XQ(B) and $(\Sigma;\sigma)$ is an environment on which e is defined, then every value in $e(\Sigma;\sigma)$ is of the form $(\Sigma; s)$. Moreover, let $(\Sigma; s)$ and $(\Sigma; s')$ be two values in $e(\Sigma;\sigma)$. Since e defines a semi-function (Proposition 5), there exists a node-renaming ρ such that $\rho(\Sigma; s) = (\Sigma; s')$. Since hence $\rho(\Sigma) = \Sigma$ and since stores are ordered, ρ must be the identity function on nodes in Σ. Hence, $s = \rho(s) = s'$, i.e., $(\Sigma; s) = (\Sigma; s')$. Thus, expressions in XQ(*concat*), when defined, evaluate to exactly one value.

Then let e_1 and e_2 be two expressions in XQ(*concat*) with the same set of free variables and let Γ be a type assignment under which e_1 and e_2 are well-defined. By our earlier observations it follows that *check*(e_1, e_2) is well-defined under Γ if, and only if, $e_1(\Sigma;\sigma) \sqsubseteq e_2(\Sigma;\sigma)$ for every $(\Sigma;\sigma) \in \Gamma$. Since we now have a reduction from the containment problem for XQ(*concat*), which is undecidable by Proposition 17, it follows that well-definedness for XQ(*concat*, *check*) is also undecidable. □

Note however that *concat* and *check* are both monotone and generic. Hence, monotonicity and genericity alone do not imply decidability. Furthermore:

Proposition 19. *The satisfiability problem for XQ(check, concat) is decidable.*

Hence, solving the satisfiability problem is not sufficient to solve well-definedness.

The core difficulty with XQ(*check*,*concat*) is that *check*'s undefinedness on a certain input depends on the whole input, and not on a local part of it. We will therefore restrict ourselves to base operations which are undefined on an input due to a local reason. We make this notion precise as follows.

Requirements

A *requirement* $\mathbf{w}$ is a tuple $(V; P_1, \ldots, P_p)$ where V is a set of nodes and the P_j are subsets of the natural numbers. Let $w = (\Sigma; s_1, \ldots, s_p)$ be a value-tuple. We say that $\mathbf{w}$ is a *requirement on w* when V is a subset of the nodes in Σ and P_j is a subset of $[1, |s_j|]$, for every $j \in [1, p]$. A value-tuple $(\Sigma'; s'_1, \ldots, s'_p)$ *satisfies* $\mathbf{w}$ on w if $(\Sigma'; s'_1, \ldots, s'_p) \sqsubseteq w$, V is a subset of the nodes in Σ', and for every $j \in [1, p]$ there exists a witness h_j for $s'_j \sqsubseteq s_j$ such that $P_j \subseteq rng(h_j)$. Note that w itself trivially satisfies $\mathbf{w}$ on w. We will denote the set of all value-tuples which satisfy $\mathbf{w}$ on w by $[\mathbf{w}, w]$.

Example 20. As an example, $(\emptyset; \{2\})$ is clearly a requirement on $(\Sigma; \langle n, a, m, a\rangle)$. Furthermore, the value $(\varnothing; \langle a\rangle)$ satisfies $(\emptyset; \{2\})$ on $(\Sigma; \langle n, a, m, a\rangle)$. Indeed, it is clear that $(\varnothing; \langle a\rangle) \sqsubseteq (\Sigma; \langle n, a, m, a\rangle)$, and that $\emptyset$ is a subset of the nodes in $\varnothing$. Moreover, the function which maps 1 to 2 is a witness of $\langle a\rangle \sqsubseteq \langle n, a, m, a\rangle$ whose range obviously includes $\{2\}$. The value $(\varnothing; \langle m, a\rangle)$ does not satisfy $(\emptyset; \{2\})$ on $(\Sigma; \langle n, a, m, a\rangle)$ however. Indeed, there exists no witness h of

$$\langle m, a\rangle \sqsubseteq \langle n, a, m, a\rangle$$

for which $\{2\} \subseteq rng(h)$. □

Locally-undefinedness

Let $R \subseteq \mathcal{V}_p \times \mathcal{V}$ be a relation and let $w \in \mathcal{V}_p$ such that $R(w) = \emptyset$. A requirement $\mathbf{w}$ on w is a *reason* why $R(w) = \emptyset$ if $R(v) = \emptyset$ for every $v \in [\mathbf{w}, w]$. Intuitively, a reason why

[8] Remember that renamings are the identity on the booleans. This explains why for example *eq* can be generic.

[9] We note that, in contrast, the corresponding problem in a set-based data model *is* decidable [10].

[10] We note that *check* is not a base operation in XQuery. Our aim here is merely to show that complex undefinedness behavior can significantly influence the well-definedness problem.

$R(w) = \emptyset$ describes a "part" of w which causes R to be undefined on w.

The *size* of a requirement $\mathbf{w} = (V; P_1, \ldots, P_p)$, denoted by $|\mathbf{w}|$, is the maximum of $|V|$, $|P_1|$, $\ldots$, $|P_p|$. We say that R is *locally-undefined* if there exists a constant k such that for every v on which R is undefined there exists a reason why $R(v) = \emptyset$ of size at most k. We call k a *witness* of the fact that R is locally-undefined. Intuitively, a locally-undefined base operation cannot base its decision to be undefined on a certain input on the whole input, but only on a small part of it.

Example 21. XQuery's children axis is locally-undefined with witness $k = 1$. Indeed, suppose that *children* is undefined on $w = (\Sigma; s)$. This only happens if there exists $j \in [1, s]$ such that $s(j)$ is a atom. Let $\mathbf{w}$ be the requirement $(\emptyset; \{j\})$ on $(\Sigma; s)$. It is clear that $\mathbf{w}$ has size one. Furthermore, let $(\Sigma'; s') \in [\mathbf{w}, w]$. Since there then exists a witness h of $s' \sqsubseteq s$ with $\{j\} \subseteq rng(h)$, there exists $i \in [1, |s'|]$ such that $h(i) = j$. It follows that s' also mentions an atom since $s'(i) = s(h(i)) = s(j)$. Hence, *children* is undefined on $(\Sigma'; s')$ and $\mathbf{w}$ is a reason why $children(w) = \emptyset$. □

We can generalize generalize Example 21 as follows:

Proposition 22. *The base operations concat, children, parent, preceding-sibling, following-sibling, descendant, ancestor, data, eq, is, ≪, is-element, is-text, node-name, content, element, text, empty, +, and × are all locally undefined.*

Note that hence locally-undefinedness alone is not powerful enough to prevent the construction of XQ(*concat*, *children*, *eq*, *node-name*, *content*, *element*, *empty*) and XQ($+, \times$), for which well-definedness is undecidable.

Unfortunately, locally-undefinedness does not transfer from base-operations to expressions, a fact which can also cause trouble:

Proposition 23. *There exists a finite set B of monotone, generic, and locally-undefined base operations for which there exist non-locally-undefined expressions in XQ(B) and for which the well-definedness problem of XQ(B) is undecidable.*

The core difficulty with this set of base operations B is that it contains a base operation which is non-local in the sense that a part of the output can not necessarily be attributed to a "small" part of the input. We will therefore restrict ourselves to base operations where every part of the output depends only on a local part of the input. We make this notion precise as follows.

Locality

We say that R is *local* if there exists a computable increasing function c mapping natural numbers to natural numbers such that for every input v, every $w \in R(v)$, and every requirement $\mathbf{w}$ on w there exists a requirement $\mathbf{v}$ on v of size at most $c(|\mathbf{w}|)$ such that for every $u \in [\mathbf{v}, v]$, if $R(u) \neq \emptyset$ then $R(u) \cap [\mathbf{w}, w] \neq \emptyset$. We call c a *witness* of the fact that R is local.

Intuitively, $\mathbf{w}$ describes part of the output of R on v. The fact that $R(u) \cap [\mathbf{w}, w] \neq \emptyset$ for every $u \in [\mathbf{v}, v]$ intuitively indicates that $\mathbf{v}$ is a part of the input v which causes $\mathbf{w}$ to be part of the output. The fact that $\mathbf{v}$ cannot be greater than $c(|\mathbf{w}|)$ indicates that $\mathbf{v}$ is local.

Example 24. The kind test *is-element* is local as witnessed by the identity function. Indeed, let $w \in is\text{-}element(v)$ for some v. Since $is\text{-}element(v)$ is hence defined, we know that v is of the form $v = (\Sigma; \langle n \rangle)$. By definition of *is-element*, $w = (\Sigma; \langle n \in \mathcal{N}^e \rangle)$. Let $\mathbf{w} = (V; P)$ be a requirement on w. Note that in particular $P \subseteq \{1\}$. Then take $\mathbf{v} = (V; \emptyset)$. It is clear that $\mathbf{v}$ is a requirement on v of size at most $|\mathbf{w}|$. Let $u \in [\mathbf{v}, v]$ and suppose that $is\text{-}element(u)$ is defined. Note that hence the sequence of u must be a singleton node. Since also $u \in [\mathbf{v}, v]$ it follows that u is of the form

$$u = (\Sigma'; \langle n \rangle)$$

with $\Sigma' \sqsubseteq \Sigma$ and V a subset of the nodes in Σ'. Then

$$is\text{-}element(u) = \{(\Sigma'; \langle n \in \mathcal{N}^e \rangle)\}.$$

Moreover, it is clear that $(\Sigma'; \langle n \in \mathcal{N}^e \rangle) \sqsubseteq (\Sigma; \langle n \in \mathcal{N}^e \rangle)$. Finally, since $P \subseteq \{1\}$, it follows that the identity function is certainly a witness of $\langle n \in \mathcal{N}^e \rangle \sqsubseteq \langle n \in \mathcal{N}^e \rangle$ whose range includes P. Hence, $is\text{-}element(u) \cap [\mathbf{w}, w] \neq \emptyset$, as desired. □

We can generalize Example 24 as follows:

Proposition 25. *The base operations concat, children, parent, preceding-sibling, following-sibling, descendant, ancestor, data, eq, is, ≪, is-element, is-text, node-name, content, element, text, empty, +, ×, and check are all local.*

Note that hence locality alone is not powerful enough to prevent the construction of XQ(*concat*, *children*, *eq*, *node-name*, *content*, *element*, *empty*), XQ($+, \times$), and XQ(*concat*, *check*), for which well-definedness is undecidable.

Locally-undefinedness transfers to expressions if B only includes monotone, local, and locally-undefined base operations:

Proposition 26. *If B is a finite set of monotone, local, and locally-undefined base operations, then every expression e in XQ(B) also defines a locally-undefined relation. Moreover, a witness of this locally-undefinedness can be effectively computed from e.*

5. DECIDABILITY RESULT

The conditions on B proposed in Section 4 are strong enough to guarantee decidability. To see why, we first introduce the following notions.

Definition 27. The *size* $|\Sigma|$ of a store Σ is the number of nodes in Σ. The *size* $|(\Sigma; s_1, \ldots, s_p)|$ of a value-tuple $(\Sigma; s_1, \ldots, s_p)$ is the sum $|\Sigma| + |s_1| + \cdots + |s_p|$.

Lemma 28. *For every type τ there exists a computable, increasing function c_τ mapping natural numbers to natural numbers such that for every $w \in \tau$ and every requirement $\mathbf{w}$ on w there exists $v \in [\mathbf{w}, w] \cap \tau$ of size at most $c_\tau(|\mathbf{w}|)$. Moreover, an arithmetic expression defining c_τ is effectively computable from τ.*

Theorem 29. *If B is a finite set of monotone, generic, local, and locally-undefined base operations then the well-definedness problem for XQ(B) is decidable.*

Proof. Suppose that $e \in \mathrm{XQ}(B)$ is not well-defined under type assignment Γ on e. Then there exists some context $w \in \Gamma$ such that $e(w) = \emptyset$. By Proposition 26 there exists a natural number k, computable from e, and a requirement $\mathbf{w}$ on w of size at most k such that $e(v) = \emptyset$ for all $v \in [\mathbf{w}, w]$. By Lemma 28 there exists $v \in [\mathbf{w}, w] \cap \Gamma$ of size at most

$$l := \sum_{x \in FV(e)} c_{\Gamma(x)}(k).$$

Here, $c_{\Gamma(x)}$ is a function for which a defining arithmetic expression is computable from $\Gamma(x)$, for every $x \in dom(\Gamma)$. Hence, l is computable from e and Γ. Note that $e(v) = \emptyset$. It is easy to see that v contains at most l nodes and that v can mention at most l different atoms. Let N be a set of nodes consisting of l element nodes and l text nodes. Let A be a set of atoms containing all constants mentioned in e and l other atoms. Then surely there exists a renaming ρ which is the identity on constants in e such that $\rho(v)$ contains only nodes in N and mentions only atoms in A. By Proposition 16, $e(\rho(v))$ is also undefined.

Hence, in order to check if e is well-defined under Γ it suffices to enumerate all contexts $v' \in \Gamma$ of size at most l with nodes in N and atoms in A, and check whether $e(v')$ is defined. There are only a finite number of such v', from which the result follows. □

Since all base operations mentioned in Section 3, except *data* and *empty*, are monotone, generic, locally-undefined and local by Propositions 14, 22, and 25, it follows in particular:

Corollary 30. *Well-definedness for XQ(concat, children, parent, preceding-sibling, following-sibling, descendant, ancestor, eq, is, ≪, node-name, content, is-element, is-text, element, text) is decidable.*

Since satisfiability reduces to well-definedness, as we have noted in Section 4.1, it follows that satisfiability for this fragment is hence also decidable.

In contrast, the *semantic type-checking problem* (i.e., is, for every input in a given input type, the output of a given expression always in a given output type) for this fragment is undecidable [3].

6. CONCLUSION

We have shown that well-definedness for $\mathrm{XQ}(B)$ is decidable if B contains only generic, monotone, local, and locally-undefined base operations. Violation of any one of these conditions allows the definition of a language for which well-definedness is undecidable.

It is clear however that the number of possible counter-examples we need to check according to decision procedure outlined in the proof of Theorem 29 can grow huge very fast. Hence the obvious question for future work: what is the computational complexity of well-definedness?

Acknowledgments

I thank Frank Neven and Jan Van den Bussche for their constructive comments on an earlier version of this paper.

7. REFERENCES

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

[2] N. Alon, T. Milo, F. Neven, D. Suciu, and V. Vianu. Typechecking XML views of relational databases. *ACM Transactions on Computational Logic*, 4(3):315–354, 2003.

[3] N. Alon, T. Milo, F. Neven, D. Suciu, and V. Vianu. XML with data values: typechecking revisited. *Journal of Computer and System Sciences*, 66(4):688–727, 2003.

[4] F. Bancilhon and S. Khoshafian. A calculus for complex objects. In *Proceedings of the fifth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 53–60. ACM Press, 1986.

[5] P. V. Biron and A. Malhotra. *XML Schema Part 2: Datatypes*. W3C Recommendation, May 2001.

[6] S. Boag, D. Chamberlin, M. F. Fernández, D. Florescu, J. Robie, and J. Siméon. *XQuery 1.0: An XML Query Language*. W3C Working Draft, February 2005.

[7] P. Buneman, M. Fernandez, and D. Suciu. UnQL: a query language and algebra for semistructured data based on structural recursion. *VLDB Journal*, 9(1):76–110, 2000.

[8] P. Buneman, S. A. Naqvi, V. Tannen, and L. Wong. Principles of programming with complex objects and collection types. *Theoretical Comput. Sci.*, 149(1):3–48, 1995.

[9] D. Chamberlin, P. Fankhauser, D. Florescu, M. Marchiori, and J. Robie. *XML Query Use Cases*. W3C Working Draft, November 2003.

[10] X. Dong, A. Halevy, and I. Tatarinov. Containment of nested XML queries. In *Proceedings of the 30th VLDB Conference*, pages 132–143. Morgan Kaufmann, 2004.

[11] D. Draper, P. Fankhauser, M. F. Fernández, A. Malhotra, K. Rose, M. Rys, J. Siméon, and P. Wadler. *XQuery 1.0 and XPath 2.0 Formal Semantics*. W3C Working Draft, February 2005.

[12] M. F. Fernández, D. Florescu, A. Levy, and D. Suciu. Declarative specification of Web sites with Strudel. *The VLDB Journal*, 9:38–55, 2000.

[13] M. F. Fernández, A. Malhotra, J. Marsh, M. Nagy, and N. Walsh. *XQuery 1.0 and XPath 2.0 Data Model*. W3C Working Draft, February 2005.

[14] J. Hidders, J. Paredaens, R. Vercammen, and S. Demeyer. A light but formal introduction to XQuery. In *Proceedings of the Second International XML Database Symposium (XSym 2004)*, volume 3186 of *Lecture Notes in Computer Science*, pages 5–20. Springer-Verlag, 2004.

[15] Y. Ioannidis and R. Ramakrishnan. Containment of conjunctive queries: Beyond relations as sets. *ACM Transactions on Database Systems*, 20(3):288–324, September 1995.

[16] H. Katz, editor. *XQuery from the Experts*. Addison-Wesley, 2003.

[17] L. Lakshmanan, G. Ramesh, H. Wang, and Z. J. Zhao. On testing satisfiability of tree pattern queries. In *Proceedings of the 30th VLDB Conference*, pages 120–131. Morgan Kaufmann, 2004.

[18] A. Malhotra, J. Melton, and N. Walsh. *XQuery 1.0 and XPath 2.0 Functions and Operators*. W3C Working Draft, February 2005.

[19] Y. Matiyasevich. *Hilbert's 10th Problem*. MIT Press, 1993.

[20] H. S. Thompson, D. Beech, M. Maloney, and N. Mendelsohn. *XML Schema Part 1: Structures*. W3C Recommendation, May 2001.

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

[22] L. Wong. *Querying nested collections*. PhD thesis, University of Pennsylvania, 1994.