Well-Definedness and Semantic Type-Checking in the Nested Relational Calculus and XQuery Extended Abstract

Jan Van den Bussche[1], Dirk Van Gucht[2,*], and Stijn Vansummeren[1,**]

[1] Limburgs Universitair Centrum, Diepenbeek, Belgium
{jan.vandenbussche, stijn.vansummeren}@luc.ac.be
[2] Indiana University, Bloomington, Indiana, USA
vgucht@cs.indiana.edu

Abstract. Two natural decision problems regarding the XML query language XQuery are well-definedness and semantic type-checking. We study these problems in the setting of a relational fragment of XQuery. We show that well-definedness and semantic type-checking are undecidable, even in the positive-existential case. Nevertheless, for a "pure" variant of XQuery, in which no identification is made between an item and the singleton containing that item, the problems become decidable. We also consider the analogous problems in the setting of the nested relational calculus.

1 Introduction

Much attention has been paid recently to XQuery, the XML query language currently under development by the World Wide Web Consortium [5, 9]. 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 [7]:

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

This expression should create for each book published by Springer-Verlag 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 first argument to the element constructor must be a singleton list.

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

* Supported by NSF Grant IIS-0082407.
** Research Assistant of the Fund for Scientific Research - Flanders (Belgium).

T. Eiter and L. Libkin (Eds.): ICDT 2005, LNCS 3363, pp. 99–113, 2005.

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 therefore is equipped with a static type system (based on XML Schema [4, 18]) 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, i.e., there are expressions which are well-defined, but not well-typed.

Can we find fragments of XQuery for which the well-definedness problem is decidable? In this paper we will study *Relational XQuery* (RX), a set-based fragment of XQuery where we omit recursive functions, only allow the child axis, take a value-based point of view (i.e., we ignore node identity), and use a type system similar to that of the nested relational or complex object data model [1, 6, 19]. We regard RX as the "first-order database fragment" of XQuery.

Even for RX, the well-definedness problem is still undecidable, due to two features which allow us to simulate the relational algebra: quantified expressions and type switches. Surprisingly, however, well-definedness remains undecidable for RX without these features, which we call positive-existential RX or PERX for short.

The core difficulty here is due to the fact that in the XQuery data model an item is identified with the singleton containing that item [11]. In a set-based model this identification becomes difficult to analyze, since $\{i, j\}$ is a singleton if and only if $i = j$. Since, as shown in the example above, there are expressions which are undefined on non-singleton inputs, this implies that in order to solve the well-definedness problem, one also needs to solve the equivalence problem. Indeed, we will see that the equivalence problem for PERX is undecidable.

Nevertheless, for a "pure" variant of PERX, in which no identification is made between an item and the singleton containing that item, well-definedness becomes decidable. We actually prove this result not for pure PERX itself, but for PENRC: the positive-existential fragment of the *nested relational calculus* [6, 19], which is well-known from the complex object data model, and whose well-definedness problem is interesting in its own right.

All our results hold not only for well-definedness, but also for *semantic type-checking*: given an expression, an input type and an output type, check whether the expression always returns outputs adhering to the output type on inputs adhering to the input type.

In the main body of the paper we will work in a set-based data model. Considering that the real XML data model is list-based, at the end of the paper we will discuss how and if our results transfer to a list-based or bag-based setting.

Related Work. The semantic type-checking problem has already been studied extensively in XML-related query languages [2, 3, 13, 14, 15, 17]. In particular, our setting closely resembles that of Alon et al. [2, 3] who, like us, study the problem in the presence of data values. In particular they have shown that (un)decidability depends on the expressiveness of both the query language and the type system. While the query language of Alon et al. can simulate PERX, our results do not follow immediately from theirs, since their type system is incompatible with ours [16].

2 Relational XQuery

In what follows we will need to define various query languages. In some definitions it will help to talk abstractly about a query language. To this end, we define a *query language* Q as a tuple $(V, T, E, [\![.]\!])$ where V is a set of *values*; T is a set of *types*; E is a set of *expressions*; and $[\![.]\!]$ is the interpretation function giving a semantics to types and expressions. The set V is also referred to as the data model.

We assume to be given an infinite set $\mathcal{X} = \{x, y, \dots\}$ of *variables*. Every expression e has associated with it a finite set $FV(e) \subseteq \mathcal{X}$ of *free variables*. An *environment* on e is a function $\sigma : FV(e) \to V$ which associates to each $x \in FV(e)$ a value $\sigma(x) \in V$. A *type assignment* on e is a function $\Gamma : FV(e) \to T$ which associates to each $x \in FV(e)$ a type $\Gamma(x) \in T$. If ρ is an environment (or a type assignment), and v is a value (respectively a type), then we write $x : v, \rho$ for the environment (respectively type assignment) ρ' with domain $dom(\rho) \cup \{x\}$ such that $\rho'(x) = v$ and $\rho'(y) = \rho(y)$ for $y \neq x$. Intuitively, environments describe the input to expressions, and type assignments describe their type.

Every type τ is associated with a set $[\![\tau]\!]$ of values. An environment σ is *compatible with* a type assignment Γ, denoted by $\sigma \in \Gamma$, if they have the same domain and $\sigma(x) \in [\![\Gamma(x)]\!]$ for all x. Every expression e has associated with it a (possibly partial) computable function $[\![e]\!]$ which associates environments on $FV(e)$ to values in V. We call $[\![e]\!]$ the *semantics* of e.

In order not to burden our notation we will identify types and expressions with their respective interpretations, and write for example $e(\sigma)$ for $[\![e]\!](\sigma)$.

2.1 Relational XQuery Data Model

In this section we define a set-based fragment of the XQuery data model [11] called the *Relational XQuery (RX) data model*. We take a value-based point of view (i.e., we ignore node identity), focus on data values, element nodes and data nodes (known as text nodes in XQuery), and abstract away from the other features in the XQuery data model such as attributes.

We assume to be given a recursively enumerable set $\mathcal{A} = \{a, b, \dots\}$ of *atoms*. An *item* is an atom or a *node*. A node is either an *element node* $\langle a : N \rangle$ or a *data node* $\langle a \rangle$, where $a \in \mathcal{A}$ and N is a finite set of nodes (N is called the content of the element node). An *RX-value*, finally, is any finite set of items. Note that, as in the XQuery data model, atoms can only occur at the "top level" of a value. Inside element nodes they must be encapsulated in a data node.

An *RX-type* τ is a term generated by the following grammar:

$$\begin{aligned} \tau &::= \mathbf{coll}(\iota) \mid \mathbf{single}(\iota) \\ \iota &::= \mathbf{atom} \mid \nu \mid \iota \cup \iota \\ \nu &::= \mathbf{data} \mid \mathbf{elem}(\gamma) \mid \nu \cup \nu \\ \gamma &::= \mathbf{coll}(\nu) \mid \mathbf{single}(\nu) \end{aligned}$$

Here, τ ranges over types, ι ranges over item types, ν ranges over node types, and γ ranges over node content types. An RX-type denotes a set of RX-values:

- **data** denotes the set of all data nodes,
- **elem**(γ) denotes the set of all element nodes $\langle a : N \rangle$ for which N is in the denotation of γ,
- **atom** denotes the set $\mathcal{A}$ of all atoms,
- $\iota_1 \cup \iota_2$ denotes the union of the denotations of ι_1 and ι_2,
- **coll**(ι) denotes the set of all finite sets over the denotation of ι, and
- **single**(ι) denotes the set of all singletons over the denotation of ι.

Note that every γ is also a τ, and hence the denotation of terms produced by γ is subsumed in the definition above.

An *RX-kind* κ is a term generated by the following grammar:

$$\kappa ::= \mathbf{atom} \mid \mathbf{data} \mid \mathbf{elem} \mid \kappa \cup \kappa$$

An RX-kind denotes a set of items, which can be the set of all atoms, the set of all data nodes, the set of all element nodes, or the union of the denotations of two kinds.

Discussion. The type system we have defined above is quite simple. Types merely indicate the many-or-one cardinality of a value, and the kinds of items that can appear in it. Only values of a fixed maximal nesting height can be described in our type system. This is justified because the expressions in the XQuery fragment RX we will work with in this paper can look only a fixed number of nesting levels down anyway. Also, it is a public secret that most XML documents in practice have nesting heights at most five or six, and that unbounded-depth nesting is not needed for many XML data processing tasks.

The presence of the **single** type constructor is justified by the fact that an item i is identified with the singleton set $\{i\}$ in the XQuery data model [11]. Consequently, an XQuery expression in which the input is always expected to be a string actually receives singleton strings as inputs. Its input type would therefore be **single**(**atom**) in our setting.

Our types also do not specify anything about the names of element nodes, but this is an omission for the sake of simplicity; we could have added node types of the form $\mathbf{elem}_a(\gamma)$, with a the atom that must be the name of the element node, without sacrificing any of the results we present in this paper.

2.2 Relational XQuery Syntax and Semantics

A *Relational XQuery expression* is an expression generated by the following grammar:

$$\begin{aligned} e ::= {} & x \\ & \mid \mathit{text}\{e\} \mid \mathit{elem}\{e\}\{e\} \mid \mathit{data}(e) \mid \mathit{name}(e) \mid \mathit{children}(e) \\ & \mid () \mid e, e \mid \mathit{for}\ x : \kappa\ \mathit{in}\ e\ \mathit{return}\ e \\ & \mid \mathit{if}\ e\ \mathit{eq}\ e\ \mathit{then}\ e\ \mathit{else}\ e \mid \mathit{if}\ e = \emptyset\ \mathit{then}\ e\ \mathit{else}\ e \mid \mathit{if}\ e \in \tau\ \mathit{then}\ e\ \mathit{else}\ e \end{aligned}$$

Here, e ranges over RX-expressions, x ranges over variables, τ ranges over RX-types and κ ranges over RX-kinds. The *free variables* of e are defined in the usual way, and will be denoted by $FV(e)$.

The semantics of RX is parameterized by two "oracle" functions:

- *content*, which maps element nodes to atoms; and
- *concat*, which maps finite sets of atoms to atoms.

We further define the following (partial) functions on values:

- $data(v) = \{a \mid a \in v\} \cup \{a \mid \langle a\rangle \in v\} \cup \{content(\langle a : N\rangle) \mid \langle a : N\rangle \in v\}$,
- $name(v)$, which is $\{a\}$ if v is a singleton element node $\{\langle a : N\rangle\}$; $concat(v)$ if v is empty; and undefined otherwise.
- $children(v)$, which is undefined if there is some atom in v, and otherwise returns
$$\bigcup\{N \mid \langle a : N\rangle \in v\}.$$
- $construct(v, w)$ which is undefined if $data(v)$ is not a singleton atom $\{a\}$; and returns $\langle a : N\rangle$ otherwise, where N is obtained from w by replacing every atom in w by a corresponding data node:
$$N = \{\langle a\rangle \mid a \in w\} \cup \{i \mid i \in w,\ i \text{ is a node}\}.$$

Let e be an RX-expression and let σ be an RX-environment on e.[1] The *semantics* $e(\sigma)$ of e under σ can now be inductively defined as follows:

$$\begin{aligned}
x(\sigma) &= \sigma(x)\\
text\{e\}(\sigma) &= \{\langle concat(data(e(\sigma)))\rangle\}\\
elem\{e_1\}\{e_2\}(\sigma) &= \{construct(e_1(\sigma), e_2(\sigma))\}\\
data(e)(\sigma) &= data(e(\sigma))\\
name(e)(\sigma) &= name(e(\sigma))\\
children(e)(\sigma) &= children(e(\sigma))\\
()(\sigma) &= \emptyset\\
e_1, e_2(\sigma) &= e_1(\sigma) \cup e_2(\sigma)\\
for\ x : \kappa\ in\ e_1\ return\ e_2 &= \bigcup\{e_2(x : \{i\}, \sigma) \mid i \in e_1(\sigma) \cap \kappa\}
\end{aligned}$$

$$(if\ e_1\ eq\ e_2\ then\ e_3\ else\ e_4)(\sigma) = \begin{cases} e_3(\sigma) & \text{if } data(e_1(\sigma)) = data(e_2(\sigma)) = \{a\},\\ & \text{with } a \text{ an atom}\\ e_4(\sigma) & \text{if } data(e_1(\sigma)) = \{a\},\\ & data(e_2(\sigma)) = \{b\},\\ & \text{with } a \text{ and } b \text{ atoms, } a \neq b \end{cases}$$

$$(if\ e_1 = \emptyset\ then\ e_2\ else\ e_3)(\sigma) = \begin{cases} e_2(\sigma) & \text{if } e_1(\sigma) = \emptyset\\ e_3(\sigma) & \text{otherwise} \end{cases}$$

$$(if\ e_1 \in \tau\ then\ e_2\ else\ e_3)(\sigma) = \begin{cases} e_2(\sigma) & \text{if } e_1(\sigma) \in \tau\\ e_3(\sigma) & \text{otherwise} \end{cases}$$

[1] Recall from the beginning of this section that σ assigns an RX-value to each free variable of e.

Note that $e(\sigma)$ is not necessarily defined: this models the situations in which XQuery expression evaluation produces a run-time error. Specifically, $e(\sigma)$ can become undefined for the following reasons:

- $e = \mathit{elem}\{e_1\}\{e_2\}$, and $\mathit{data}(e_1(\sigma))$ is not a singleton atom. (This can only happen if $e_1(\sigma)$ is not a singleton.)
- $e = \mathit{name}(e')$, and $e'(\sigma)$ is not the empty set, or not a singleton element node.
- $e = \mathit{children}(e')$, and $e'(\sigma)$ contains an atom.
- $e = \mathit{if}\ e_1\ \mathit{eq}\ e_2\ \mathit{then}\ e_3\ \mathit{else}\ e_4$, and $\mathit{data}(e_1(\sigma))$ is not a singleton atom, or $\mathit{data}(e_2(\sigma))$ is not a singleton atom. (This can only happen if $e_1(\sigma)$ respectively $e_2(\sigma)$ is not a singleton.)

Relation to XQuery. The RX query language corresponds to a set-based version of XQuery [5, 9] where we have omitted recursive functions, literals, arithmetic expressions, generalized and order comparisons, and only allow the children axis. We have replaced XQuery quantified expressions by the emptiness test (which is equivalent in expressive power), and have moved kind tests from XQuery step expressions to the "for" expression. As an example, the XQuery step expression $\$x/\mathit{child} :: \mathit{text}()$ can be expressed in RX as

$$\mathit{for}\ z : \mathbf{data}\ \mathit{in}\ \mathit{children}(x)\ \mathit{return}\ z.$$

The "oracle" functions *concat* and *content* model features which are present in XQuery, but which are clumsy to take into account in our data model. For example *name* applied to the empty sequence returns the empty string in XQuery. Furthermore, applying *data* to a singleton element node in XQuery returns the "string content" of the node. This is (roughly speaking) a concatenation of all atoms (converted to strings) encountered in a depth-first left-to-right traversal of the node's content.

3 Well-Definedness and Semantic Type-Checking

As we have noted in Section 2.2, the semantics $e(\sigma)$ of RX-expression e under environment σ can be undefined. This leads us to the following definition.

Definition 1. *The well-definedness problem for a query language Q consist of checking, given a Q-expression e and a Q-type assignment Γ on e: whether $e(\sigma)$ is defined for every $\sigma \in \Gamma$. In this case we say that e is well-defined under Γ.*

A problem which is related to well-definedness is the *semantic type-checking* problem:

Definition 2. *The* semantic type-checking *problem for a query language Q consist of checking, given a Q-expression e, a Q-type assignment Γ on e such that e is well-defined under Γ, and a Q-type τ: whether $e(\sigma) \in \tau$ for every $\sigma \in \Gamma$. In this case we say that τ is an output type for e under Γ.*

4 Undecidability Results

We will show that well-definedness for RX is undecidable, even for a quite restricted fragment. Our results do not depend on the particular interpretation given to the oracle functions *concat* and *content*.

Let us begin by defining RX^- as the fragment of RX where

- we disallow data node construction expressions of the form $\mathit{text}\{e\}$;
- we disallow data extraction expressions of the form $\mathit{data}(e)$; and
- we disallow kind tests, or equivalently, we only allow the use of the single "universal" kind $\mathbf{atom} \cup \mathbf{data} \cup \mathbf{elem}$.

An RX^--expression e is *positive existential* if it does not contain *emptiness tests* of the form $\mathit{if}\ e_1 = \emptyset\ \mathit{then}\ e_2\ \mathit{else}\ e_3$, or *type switches* of the form $\mathit{if}\ e_1 \in \tau\ \mathit{then}\ e_2\ \mathit{else}\ e_3$. We denote the language of all positive-existential RX^- expressions by PERX^-, and we will mention specific features added back to PERX^- in square brackets. Thus, PERX^-[empty] includes emptiness tests, and type switches are included in PERX^-[type].

Proposition 1. *Type switches can be used to simulate emptiness tests, i.e. PERX*$^-$*[empty] is a semantic subset of PERX*$^-$*[type].*

Indeed, $\mathit{if}\ e_1 = \emptyset\ \mathit{then}\ e_2\ \mathit{else}\ e_3$ can be expressed as follows:

$$\mathit{if}\ (\mathit{for}\ x\ \mathit{in}\ e_1\ \mathit{return}\ \mathit{elem}\{a\}\{()\}) \in \mathbf{coll}(\mathbf{data})\ \mathit{then}\ e_2\ \mathit{else}\ e_3$$

The following proposition is not surprising, and parallels earlier results on semistructured query languages such as StruQL [10]:

Proposition 2. *PERX*$^-$*[empty] can simulate the relational algebra. Concretely, for every relational algebra expression ϕ over database schema $\mathbf{S}$, there exists a PERX*$^-$*[empty]-expression e_ϕ and a type assignment $\Gamma_{\mathbf{S}}$, such that*

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

Consequently, satisfiability (i.e., nonempty output on at least one input) is undecidable for PERX^-[empty] (and thus for RX^-), because it is undecidable for the relational algebra. Since the expression

$$\mathit{for}\ x\ \mathit{in}\ e\ \mathit{return}\ \mathit{elem}\{()\}\{()\}$$

is well-defined if, and only if, e is unsatisfiable, we obtain:

Corollary 1. *Well-definedness for PERX*$^-$*[empty] (and thus RX) is undecidable.*

What is perhaps more surprising is that without emptiness test, we remain undecidable:

Theorem 1. *Well-definedness for PERX$^-$ is undecidable.*

Proof (Crux). The proof goes by reduction from the implication problem for functional and inclusion dependencies, which is known to be undecidable [1, 8].

Let Σ be a set of functional and inclusion dependencies, and let ρ be an inclusion dependency. We show in the full version of this paper that we can construct two expressions e_1 and e_2, a type assignment Γ and a node content type γ, such that

- e_1 and e_2 are well-defined under Γ,
- γ is an output type for e_1 and e_2 under Γ, and,
- $e_1(\sigma) = e_2(\sigma)$ for every $\sigma \in \Gamma$ if, and only if, ρ is implied by Σ.

Consequently, the expression $\mathit{name}(\mathit{elem}\{a\}\{e_1\}, \mathit{elem}\{a\}\{e_2\})$ is well-defined under Γ if, and only if, ρ is implied by Σ. □

As a corollary to the proof, we note:

Corollary 2. *Equivalence of PERX$^-$ expressions is undecidable.*

We further derive:

Corollary 3. *Semantic type-checking for PERX$^-$ is undecidable.*

Indeed, referring to the above proof sketch of Theorem 1, e_1 and e_2 are equivalent if, and only if, $(\mathit{elem}\{a\}\{e_1\}, \mathit{elem}\{a\}\{e_2\})$ has output type $\mathbf{single}(\mathbf{elem}(\gamma))$.

We remark that to establish undecidability of well-definedness we do not need singleton types. For undecidability of semantic type-checking, we do.

5 Pure RX

In the XQuery data model, an item i is identified with the singleton $\{i\}$ [11]. With this identification, it is indeed natural to let, e.g., $\mathit{name}(e)$ be undefined when $e(\sigma)$ is a set with more than one element. As we have seen in the previous Section, it is exactly this behavior that causes well-definedness to be undecidable.

So let us define a version of RX, called *pure RX*, which does not explicitly identify an item i with $\{i\}$. We will show in Section 6 that well-definedness and semantic type-checking for the positive-existential fragment of pure RX is decidable.

A *pure RX-value* is an item or a set of items. A *pure RX-type* τ is a term generated by the following grammar:

$$\tau ::= \mathbf{coll}(\iota) \mid \iota \mid \tau \cup \tau$$
$$\iota ::= \mathbf{atom} \mid \nu \mid \iota \cup \iota$$
$$\nu ::= \mathbf{data} \mid \mathbf{elem}(\nu_1 \cup \cdots \cup \nu_k)$$

Here, τ ranges over types, ι ranges over item types, ν ranges over node types, and $k \geq 0$.

A *pure RX-type* denotes a set of pure RX-values:

- **data** denotes the set of all data nodes,
- **elem**$(\nu_1 \cup \cdots \cup \nu_k)$ denotes the set of all element nodes $\langle a : N\rangle$ for which N is a finite set over the union of the denotations of $\nu_1, \ldots, \nu_k$.
- **atom** denotes the set $\mathcal{A}$ of all atoms,
- $\tau_1 \cup \tau_2$ denotes the union of the denotations of τ_1 and τ_2, and,
- **coll**(ι) denotes the set of all finite sets over the denotation of ι.

Note that since every ι is also a τ, the denotation of $\iota_1 \cup \iota_2$ is subsumed by the definition above.

The syntax of *pure RX* is obtained from the syntax of RX by adding a singleton constructor expression (e), and by replacing RX-types in type switch expressions by pure RX types.

In order to give the semantics of pure RX, we define the following (partial) functions on pure RX-values.

- $\mathit{data}'(v) = \{a \mid a \in v\} \cup \{a \mid \langle a\rangle \in v\}$
- $\mathit{name}'(v)$, which is a if v is an element node $\langle a : N\rangle$, and is undefined otherwise.
- $\mathit{children}'(v)$, which is undefined if there is some atom in v, and otherwise returns
$$\bigcup\{N \mid \langle a : N\rangle \in v\}.$$
- $\mathit{construct}'(v, w)$ which is undefined if v is not an atom, and returns $\langle v : N\rangle$ otherwise where N is obtained from w by replacing every atom in w by a corresponding data node:
$$N = \{\langle a\rangle \mid a \in w\} \cup \{i \mid i \in w, i \text{ is a node}\}$$

The semantics of pure RX is then defined as follows:

$$\begin{aligned}
x(\sigma) &= \sigma(x)\\
\mathit{text}\{e\}(\sigma) &= \langle a\rangle \quad \text{if } e(\sigma) = a\\
\mathit{elem}\{e_1\}\{e_2\}(\sigma) &= \mathit{construct}'(e_1(\sigma), e_2(\sigma))\\
\mathit{data}(e)(\sigma) &= \mathit{data}'(e(\sigma))\\
\mathit{name}(e)(\sigma) &= \mathit{name}'(e(\sigma))\\
\mathit{children}(e)(\sigma) &= \mathit{children}'(e(\sigma))\\
()(\sigma) &= \emptyset\\
(e)(\sigma) &= \{e(\sigma)\} \quad \text{if } e(\sigma) \text{ is an item}\\
e_1, e_2(\sigma) &= e_1(\sigma) \cup e_2(\sigma)\\
\mathit{for}\ x : \kappa\ \mathit{in}\ e_1\ \mathit{return}\ e_2 &= \bigcup\{e_2(x : i, \sigma) \mid i \in e_1(\sigma) \cap \kappa\}
\end{aligned}$$

$$(\mathit{if}\ e_1\ \mathit{eq}\ e_2\ \mathit{then}\ e_3\ \mathit{else}\ e_4)(\sigma) = \begin{cases} e_3(\sigma) & \text{if } e_1(\sigma), e_2(\sigma) \in \mathcal{A} \text{ and } e_1(\sigma) = e_2(\sigma)\\ e_4(\sigma) & \text{if } e_1(\sigma), e_2(\sigma) \in \mathcal{A} \text{ and } e_1(\sigma) \neq e_2(\sigma)\end{cases}$$

$$(\mathit{if}\ e_1 = \emptyset\ \mathit{then}\ e_2\ \mathit{else}\ e_3)(\sigma) = \begin{cases} e_2(\sigma) & \text{if } e_1(\sigma) = \emptyset\\ e_3(\sigma) & \text{otherwise}\end{cases}$$

$$(\mathit{if}\ e_1 \in \tau\ \mathit{then}\ e_2\ \mathit{else}\ e_3)(\sigma) = \begin{cases} e_2(\sigma) & \text{if } e_1(\sigma) \in \tau \\ e_3(\sigma) & \text{otherwise} \end{cases}$$

Note that again $e(\sigma)$ is not necessarily defined. Specifically, $e(\sigma)$ can become undefined for the following reasons:

- $e = \mathit{text}\{e'\}$, and $e'(\sigma)$ is not an atom,
- $e = \mathit{elem}\{e_1\}\{e_2\}$, and $e_1(\sigma)$ is not an atom,
- $e = \mathit{name}(e')$, and $e'(\sigma)$ is not an element node,
- $e = \mathit{children}(e')$, and $e'(\sigma)$ contains an atom,
- $e = (e')$, and $e'(\sigma)$ is not an item,
- $e = e_1, e_2$, and $e_1(\sigma)$ is not a set or $e_2(\sigma)$ is not a set,
- $e = \mathit{for}\ x : \kappa\ \mathit{in}\ e_1\ \mathit{return}\ e_2$, and $e_1(\sigma)$ is not a set or $e_2(x : i, \sigma)$ is not a set for some $i \in e_1(\sigma) \cap \kappa$, or,
- $e = \mathit{if}\ e_1\ \mathit{eq}\ e_2\ \mathit{then}\ e_3\ \mathit{else}\ e_4$, and $e_1(\sigma)$ or $e_2(\sigma)$ is not an atom.

Pure PERX

Well-definedness and semantic type-checking for the entire pure RX remains undecidable due to the presence of the emptiness test and type switch expressions. Let us define *pure PERX* as the fragment of pure RX in which these expressions are disallowed.

6 Decidability Results

In this section we will show that well-definedness and semantic type-checking for pure PERX are decidable. In fact, we will solve the corresponding problems for the nested relational calculus (NRC): the well-known standard query language for nested relations and complex objects. Indeed, this language remains fundamental and its study remains interesting in its own right. As we will see, pure PERX can be simulated by the positive-existential fragment of NRC (extended with kind-tests).

6.1 Nested Relational Calculus

An *NRC-value* is either an atom, a pair of NRC-values, or a finite set of NRC-values. Note that we allow sets to be heterogeneous. If $v = (v_1, v_2)$, then we write $\pi_1(v)$ for v_1 and $\pi_2(v)$ for v_2.

An *NRC-type* τ is a term generated by the following grammar:

$$\tau ::= \emptyset \mid \mathbf{atom} \mid \tau \times \tau \mid \tau \cup \tau \mid \mathbf{coll}(\tau)$$

An NRC-type *denotes* a set of NRC-values:

- $\emptyset$ denotes the empty set,
- **atom** denotes the set $\mathcal{A}$ of all atoms,
- $\tau_1 \times \tau_2$ denotes the cartesian product of the denotations of τ_1 and τ_2,

– $\tau_1 \cup \tau_2$ denotes the union of the denotations of τ_1 and τ_2, and,
– $\mathbf{coll}(\tau)$ denotes the set of all finite sets over the denotation of τ.

An *NRC-kind* κ is a term generated by the following grammar:

$$\kappa ::= \mathbf{atom} \mid \mathbf{coll} \mid \kappa \times \kappa \mid \kappa \cup \kappa$$

An NRC-kind denotes a set of NRC-values, which can be the set of all atoms, the set of all finite sets of values, the cartesian product of the denotation of two kinds, or the union of the denotation of two kinds.

The *positive existential nested relational calculus* (PENRC) is the set of all expressions generated by the following grammar:

$$\begin{aligned} e ::= & \; x \\ \mid & \; (e,e) \mid \pi_1(e) \mid \pi_2(e) \\ \mid & \; \emptyset \mid \{e\} \mid e \cup e \mid \bigcup e \mid \{e \mid x \in e\} \\ \mid & \; e = e \; ? \; e \; : \; e \end{aligned}$$

Here e ranges over expressions, and x ranges over variables. The PENRC with kind tests, denoted by PENRC[kind] is the PENRC extended with one additional expression:

$$e ::= \cdots \mid e \in \kappa \; ? \; e \; : \; e$$

Here, κ ranges over NRC kinds. The *free variables* of e are defined in the usual way, and will be denoted by $FV(e)$.

If e is a PENRC[kind]-expression and σ is an NRC-environment on e, then the *semantics* $e(\sigma)$ of e under σ is inductively defined as follows:

$$\begin{aligned} x(\sigma) &= \sigma(x) \\ (e_1,e_2)(\sigma) &= (e_1(\sigma), e_2(\sigma)) \\ \pi_1(e)(\sigma) &= \pi_1(e(\sigma)) \\ \pi_2(e)(\sigma) &= \pi_2(e(\sigma)) \\ \emptyset(\sigma) &= \emptyset \\ \{e\}(\sigma) &= \{e(\sigma)\} \\ (e_1 \cup e_2)(\sigma) &= e_1(\sigma) \cup e_2(\sigma) \\ (\bigcup e)(\sigma) &= \bigcup e(\sigma) \\ \{e_2 \mid x \in e_1\}(\sigma) &= \{e_2(x:v,\sigma) \mid v \in e_1(\sigma)\} \\ (e_1 = e_2 \; ? \; e_3 \; : \; e_4)(\sigma) &= \begin{cases} e_3(\sigma) & \text{if } e_1(\sigma), e_2(\sigma) \in \mathcal{A} \text{ and } e_1(\sigma) = e_2(\sigma) \\ e_4(\sigma) & \text{if } e_1(\sigma), e_2(\sigma) \in \mathcal{A} \text{ and } e_1(\sigma) \neq e_2(\sigma) \end{cases} \\ (e_1 \in \kappa \; ? \; e_2 \; : \; e_3)(\sigma) &= \begin{cases} e_2(\sigma) & \text{if } e_1(\sigma) \in \kappa \\ e_3(\sigma) & \text{otherwise} \end{cases} \end{aligned}$$

Note that $e(\sigma)$ can be undefined. For example $\pi_1(x)(\sigma)$ is undefined when $\sigma(x)$ is not a pair, and $(x \cup y)(\sigma)$ is undefined when $\sigma(x)$ is not a set. Hence, we can also study the well-definedness problem for PENRC[kind].

It is easy to see that well-definedness for full NRC: PENRC extended with an emptiness test, is undecidable. Indeed, it is well known that NRC can simulate the relational algebra [6].

6.2 Simulating RX in NRC

Formally, a *simulation* of a query language Q in a query language Q' is a function $enc : V_Q \to V_{Q'}$ such that

- for every type $\tau \in T_Q$ there exists a type $\tau' \in T_{Q'}$ such that $v \in \tau$ if and only if $enc(v) \in \tau'$, and
- for every expression $e \in E_Q$ there exists an expression $e' \in E_{Q'}$ such that
  1. $e(\sigma)$ is defined if and only if $e'(enc(\sigma))$ is defined, and
  2. if $e(\sigma)$ is defined, then $enc(e(\sigma)) = e'(enc(\sigma))$.

A simulation is *effective* if τ' can be computed from τ and e' can be computed from e.

Lemma 1. *Pure PERX can be effectively simulated in PENRC[kind].*

Proof (Crux). Consider the encoding function *enc* for which

$$enc(a) = a \qquad enc(\langle a \rangle) = ((a, a), \emptyset)$$
$$enc(\langle a : N \rangle) = (a, enc(N)) \qquad enc(v) = \{enc(i) \mid i \in v\}$$

Then *enc* is an effective simulation. It is easy to find τ' by induction on τ. Furthermore, e' can be constructed by induction on e. To illustrate this, let us write $e_1 \in \kappa \to e_2$ for $e_1 \in \kappa \ ?\ e_2 \ :\ \pi_1(\emptyset)$. Intuitively, this expression will be used to verify that the input to e' is an encoding of a legal input to e. Otherwise, we become undefined.

We can now for example simulate $text\{e\}$ by $e' \in \mathbf{atom} \to ((e', e'), \emptyset)$. We can simulate $elem\{e_1\}\{e_2\}$ by

$$e'_1 \in \mathbf{atom} \to (e'_1, \{x \in \mathbf{atom}\ ?\ ((x, x), \emptyset)\ :\ x \mid x \in e'_2\}).$$

And we can simulate $children(e)$ by $\bigcup\{\pi_2(x) \mid x \in e'\}$. □

Corollary 4. *If the well-definedness or semantic type-checking problem is decidable for PENRC[kind], then it is also decidable for pure PERX.*

6.3 Well-Definedness for PENRC[kind]

Consider the following expression:

$$e = \{\{z = y\ ?\ \pi_1(z)\ :\ y \mid y \in x\}) \mid x \in R\},$$

and let the environment σ be defined by

$$\sigma(R) = \{\{a, b\}, \{c\}, \{a, b, d\}\} \qquad \sigma(z) = d.$$

Since there is a set in $\sigma(R)$ which contains $\sigma(z)$, we will need to evaluate $\pi_1(\sigma(z))$ at some point, which is undefined. Hence, $e(\sigma)$ is undefined. Note that we do not need all elements in $\sigma(R)$ to reach the state where $e(\sigma)$ becomes undefined. Indeed, e is also undefined on the small environment σ' where $\sigma'(R) = \{\{d\}\}$ and $\sigma'(z) = d$.

We generalize this observation in the following general property. Here, we say that an environment σ is in the set $\mathcal{E}_k$ if every set occurring in $\sigma(x)$ has cardinality at most k for every $x \in dom(\sigma)$.

Lemma 2 (Small Model Property for Undefinedness). *Let e be a positive existential NRC[kind] expression, let Γ be a type assignment on e, and let σ be an environment compatible with Γ such that $e(\sigma)$ is undefined. There exists a natural number l which can be computed from e alone, and an environment $\sigma' \in \mathcal{E}_l$ compatible with Γ, such that $e(\sigma')$ is also undefined.*

We obtain:

Corollary 5. *The well-definedness problem for PENRC[kind] is decidable.*

Indeed, up to isomorphism (and expressions cannot distinguish isomorphic inputs) there are only a finite number of different input environments in $\mathcal{E}_l$ compatible with Γ. So we can test them all to see if there is a counterexample to well-definedness.

Also for semantic type-checking we have:

Lemma 3 (Small Model Property for Semantic Type-Checking). *Let e be a PENRC[kind] expression, let Γ be a type assignment on e such that e is well-defined under Γ, and let τ be a type. Let σ be an environment compatible with Γ such that $e(\sigma) \notin \tau$. There exists a natural number l which can be computed from e and τ alone, and an environment $\sigma' \in \mathcal{E}_l$ compatible with Γ, such that also $e(\sigma') \notin \tau$.*

Corollary 6. *Semantic type-checking for PENRC[kind] is decidable.*

6.4 Equivalence and Satisfiability

The above decidability results are quite sharp, because *equivalence* of PENRC expressions is undecidable. This can be proven in a similar way as Theorem 1. Of course, containment is then also undecidable. Levy and Suciu [12] have shown that a "deep" form of containment (known as simulation) is decidable for PENRC.

Another important problem is satisfiability. For example, the XQuery type system generates a type error whenever it can deduce that an expression which is not the empty set expression itself always returns the empty set. As noted in Section 4, satisfiability is undecidable for PERX$^-$[empty]. For pure PERX, and PENRC[kind], however, satisfiability can be solved using the small model

property for semantic type-checking. Indeed, a PENRC[kind] expression e is unsatisfiable under Γ if, and only if, $\mathbf{coll}(\emptyset)$ is an output type for e under Γ. We point out that, at least for PENRC without union and kind-tests, decidability of satisfiability already follows from the work of Levy and Suciu cited above.

7 Lists and Bags

In this paper we have focused our attention on a set-based abstraction of XQuery. The actual data model of XQuery is list-based however, and hence it is natural to ask how our results transfer to such a setting.

Let us denote by $\mathrm{RX}^{\mathrm{list}}$ the list-based version of RX, which can be obtained from RX as follows. The list-based RX data model is obtained by replacing "set" in the definition of the RX data model by "list". The list-based semantics of an expression is obtained from the set-based semantics by replacing every set operator by the corresponding list operator (i.e., empty set becomes empty list, union becomes concatenation, and so on). We can similarly define the bag-based version of RX, which we will denote by $\mathrm{RX}^{\mathrm{bag}}$.

We can still simulate the relational algebra in the list- and bag-based versions of PERX^-[empty] and PERX^-[type]. Hence, well-definedness and semantic type-checking for these languages is undecidable. It is an open problem however whether well-definedness and semantic type-checking in the list- and bag-based versions of PERX^- remains undecidable. Indeed, our undecidability proof depends heavily on the fact that set union is idempotent, which is not the case for list concatenation and bag union.

We can also consider a list-based and bag-based version of PENRC[kind], to which our decidability results transfer. Hence, well-definedness and semantic type-checking are decidable for pure $\mathrm{PERX}^{\mathrm{list}}$ and pure $\mathrm{PERX}^{\mathrm{bag}}$.

References

1. Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations Of Databases*. Addison-Wesley, 1995.
2. Noga Alon, Tova Milo, Frank Neven, Dan Suciu, and Victor Vianu. Typechecking XML views of relational databases. *ACM Transactions on Computational Logic*, 4(3):315–354, 2003.
3. Noga Alon, Tova Milo, Frank Neven, Dan Suciu, and Victor Vianu. XML with data values: typechecking revisited. *Journal of Computer and System Sciences*, 66(4):688–727, 2003.
4. Paul V. Biron and Ashok Malhotra. *XML Schema Part 2: Datatypes*. W3C Recommendation, May 2001.
5. Scott Boag, Don Chamberlin, Mary F. Fernández, Daniela Florescu, Jonathan Robie, and Jérôme Siméon. *XQuery 1.0: An XML Query Language*. W3C Working Draft, November 2003.
6. Peter Buneman, Shamim A. Naqvi, Val Tannen, and Limsoon Wong. Principles of programming with complex objects and collection types. *Theoretical Computer Science*, 149(1):3–48, 1995.

7. Don Chamberlin, Peter Fankhauser, Daniela Florescu, Massimo Marchiori, and Jonathan Robie. *XML Query Use Cases*. W3C Working Draft, November 2003.
8. Ashok K. Chandra and Moshe Y. Vardi. The implication problem for functional and inclusion dependencies is undecidable. *SIAM Journal on Computing*, 14(3):671–677, 1985.
9. Denise Draper, Peter Fankhauser, Mary F. Fernández, Ashok Malhotra, Kristoffer Rose, Michael Rys, Jérôme Siméon, and Philip Wadler. *XQuery 1.0 and XPath 2.0 Formal Semantics*. W3C Working Draft, February 2004.
10. Mary F. Fernández, Daniela Florescu, Alon Levy, and Dan Suciu. Declarative specification of Web sites with Strudel. *The VLDB Journal*, 9:38–55, 2000.
11. Mary F. Fernández, Ashok Malhotra, Jonathan Marsh, Marton Nagy, and Norman Walsh. *XQuery 1.0 and XPath 2.0 Data Model*. W3C Working Draft, November 2003.
12. Alon Y. Levy and Dan Suciu. Deciding containment for queries with complex objects (extended abstract). In *Proceedings of the Sixteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of Database Systems*, pages 20–31. ACM Press, 1997.
13. Wim Martens and Frank Neven. Typechecking top-down uniform unranked tree transducers. In *Database Theory - ICDT 2003*, volume 2572 of *Lecture Notes in Computer Science*, pages 64–78. Springer-Verlag, 2003.
14. Wim Martens and Frank Neven. Frontiers of tractability for typechecking simple xml transformations. In *Proceedings of the Twenty-third ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 23–34. ACM Press, 2004.
15. Tova Milo, Dan Suciu, and Victor Vianu. Typechecking for XML transformers. In *Proceedings of the Nineteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 11–22. ACM Press, 2000.
16. Frank Neven. Personal communication, May 2004.
17. Dan Suciu. Typechecking for semistructured data. In *Database Programming Languages, 8th International Workshop, DBPL 2001, Revised Papers*, volume 2397 of *Lecture Notes in Computer Science*, pages 1–20. Springer-Verlag, 2001.
18. Henry S. Thompson, David Beech, Murray Maloney, and Noah Mendelsohn. *XML Schema Part 1: Structures*. W3C Recommendation, May 2001.
19. Limsoon Wong. *Querying nested collections*. PhD thesis, University of Pennsylvania, 1994.