A Better Semantics for XQuery with Side-Effects

Giorgio Ghelli[1], Nicola Onose[2], Kristoffer Rose[3], and Jérôme Siméon[3]

[1] Università di Pisa, ghelli@di.unipi.it
[2] University of California, San Diego, nicola@cs.ucsd.edu
[3] IBM T. J. Watson Research Center, krisrose/simeon@us.ibm.com

Abstract. Formal semantics for XQuery with side-effects have been proposed in [13, 16]. We propose a different semantics which is better suited for database compilation. We substantiate this claim by formalizing the compilation of XQuery extended with updates into a database algebra. We prove the correctness of the proposed compilation by mapping both the source language and the algebra to a common core language with list comprehensions and extensible tuples.

1 Introduction

Two semantics of XQuery. The use of list comprehensions to formalize database languages has been popular since the work of Trinder and Wadler [24], and that of Buneman et al [23, 3]. More recently, the same approach has been used successfully by Fernandez et al. [11] to specify the semantics of XQuery. It notably relies on a notion of *normalization* that is now part of the XQuery Formal Semantics [7]. For instance, the following FLWOR expression applied to a variable `$d` containing the element `<doc><a>1</a><a>2</a></doc>`

```
for $x in $d/a
for $y in $d/a
where $x = $y
return ($x+$y)
```

is defined as being equivalent to the following expression in the target fragment of XQuery called the XQuery *core* (we use a different font for the core)

```
for $x in $d/a return
  for $y in $d/a return
    if ($x = $y) then ($x+$y) else ()
```

This approach has the benefit of relying on a small set of simple primitives which are well understood from functional programming, and support laws useful for optimization.

Because of the importance of FLWOR expressions as a database primitive, most compilers do not rely on normalization, but instead compile into tuple-based algebras that support traditional database optimizations [19, 18, 21]. For instance, the above query is compiled in the following plan using the algebra of [21].

```
Map{#x+#y}
 (Select{#x = #y}
  (MapConcat
   {Map{[y:ID]}(TreeJoin[a](#d))}
   (Map{[x:ID]}(TreeJoin[a](#d)))))
```

This plan manipulates streams of tuples whose fields correspond to the variables in the source code. It first builds a stream of tuples [x:v], by applying tuple construction [x:ID] to each node resulting from the navigation TreeJoin[a](#d) (i.e, $\$d/a$). It then concatenates each [x:v] tuple which each of the [y:v] tuples built by the Map{[y:ID]}(TreeJoin[a](#d)) subplan, selects the tuples that satisfy #x = #y, and computes #x+#y once for each tuple. Algebraic equivalences can be applied to that plan to introduce a more efficient join operator.

Several extensions to XQuery involving side-effects have recently been proposed [6, 13, 4] by the research community, and are being considered by the W3C [5]. While the first proposals for side-effects relied on whole-program snapshot semantics [6], meaning that a piece of code could not observe its own effect, some consensus is emerging that allowing side-effects to be visible is a key feature for new XML applications [13, 4, 16, 5]. Normalization and algebraic compilation coincide for a "pure" language such as XQuery 1.0. Unfortunately, in presence of visible side-effects, tuple based compilation and nested-for semantics diverge. Consider the following query, where `insert` exemplifies any visible side-effect.

```
for $x in $d/a
for $y in $d/a
where $x = $y
return (do insert <a>{$x+$y}</a> into $d)
```

The normalization approach, as used in [16, 13], defines the query to be equivalent to the following core expression.

```
for $x in $d/a return
 for $y in $d/a return
  if ($x = $y) then (do insert <a>{$x+$y}</a> into $d) else ()
```

Hence, it first executes the internal **for** *$y*... with *$x* bound to the first a element, and then executes the same expression with *$x* bound to the second a element. The first iteration inserts a new element a into *$d*. Hence, in the second iteration *$y* is bound to a different set of nodes. Instead, the tuple-building phase is protected from the effects of the `return` clause when the algebraic compilation is applied, as follows.

```
Map{Insert(<a>{#x+#y}</a>,#d)}
 (Select{#x = #y}
  (MapConcat
   {Map{[x:ID]}(TreeJoin[a](#d))}
   (Map{[y:ID]}(TreeJoin[a](#d)))))
```

One may argue which of the two semantics is better for the programmer. We observe that the informal, but normative, semantics for XQuery is actually defined in terms of a tuple stream, and mandates that the **return** clause is executed after the tuple-stream has been built and filtered [2]. We believe that the tuple-stream semantics is at least as natural as the nested-for semantics, and is better suited for optimization: in this example, the nested-for semantics requires the internal loop to be run on two different sequences, hence makes it impossible to use a join, while the tuple-stream semantics enables join optimization, which would produce the following plan.

```
Map{Insert(<a>{#x+#y}</a>,#d)}
 (Join{#x = #y}
   (Map{[x:ID]}(TreeJoin[a](#d)),
    Map{[y:ID]}(TreeJoin[a](#d))))
```

Scope of the paper. Our goal is to provide a list-comprehension based semantics for XQuery which enables the use of traditional tuple-based algebras for compilation and optimization. To this aim, we first formalize the tuple-stream semantics, by translating XQuery with updates into a target language with records which we call the *XQueryU core with tuples*. The result is still a nesting of for-expressions, which, in this case, are used to first build the tuple-stream, and to finally apply the return clause. For example, the code above is translated as follows, where *$environment* is a tuple-stream that only contains one context-tuple, [d:<doc><a>1<a>2</doc>].

```
for $t5 in
    for $t4 in
        for $t2 in
            (for $t0 in $environment
             return for $t1 in $d/a return [x:$t1]++ $t0)
        return for $t3 in $d/a return [y:$t3]++ $t2
    return if ($t4.x = $t4.y) then $t4 else ()
return (do insert element 'a' {$t5.x+$t5.y} into $t5.d)
```

The translation transforms variables into tuple fields, moves the outermost iteration in the innermost position and, most importantly, moves the filtering, reordering, and **return** phases at the end of the whole process, as required by the tuple-stream semantics. This formalization is not only a first step to our main theorem about the soundness of a database-like interpretation of tuple-stream semantics, but it also shows that the tuple-stream semantics admits, in a measure, the same advantages of the nested-for semantics, namely:

1. it admits a simple PL-style implementation, based on in-memory nested loops, with no need of going through algebraic compilation, for applications where a PL-style implementation may be useful.
2. it can be mapped down to a simple core language which is best suited for studies about semantics and types, as we will do in this paper.

We then formally define a database algebra for this language and a compilation function from XQuery with side-effects to the algebra, and we prove that the compilation implements the tuple-stream semantics. The target language and the algebra are typed, by a type system that keeps track of record access and concatenation. These type systems involve record concatenation and subtyping, which means that we have to address the well-known problem that these two mechanisms are incompatible for simple record types [14]. We have chosen an original solution for this old problem, based on *linearity* conditions which identify a sub-language where concatenation and simple subtyping safely coexist, and we show that this sub-language is indeed sufficient to interpret XQuery.

Related work. Design and semantics for database query languages based on comprehension were first introduced by [3, 23, 22]. This work notably led to the development of the Kleisli system which is probably the most advanced query language compiler based on functional techniques [25, 8]. However, join optimization in Kleisli is not handled at the comprehension level, but remains internal to the compiler. So far, functional optimizations did not catch on as most database management system optimizers to this day rely on tuple-based algebras [1, 20, 19]. The recent emergence of languages which blend database and programming languages features [17, 12, 4, 15] has created renewed interest in list comprehensions. We believe our work is the first to provide a list comprehension treatment of a tuple-based database algebra. The reconciliation of record subtyping with record concatenation has been the subject of a huge body of work [14]. The proposed approaches were mostly based on the addition of information about missing fields, or on the substitution of subtyping with parametric polymorphism. We instead use the simplest form of record types, but impose a linearity constraint on the code. Finally, the topic of optimization in the presence of side effects have received almost no attention so far. On notably exception is the work by Fegaras [9] which also relies on a monadic approach.

We first introduce the XQuery core with tuples (Section 2). We then introduce the source language XQueryU (Section 3) and its semantics, through a translation to the core. We finally define the typed second-order algebra (Section 4), the compilation of XQueryU into the algebra, and prove its correctness.

2 XQueryU core with tuples

In this section we define a core language with support for tuples, which we use to specify the semantics of both XQuery with updates and the corresponding algebra. This core language is based on the W3C XQuery core defined in [7].

Definition 2.1 (core syntax). Our core language has the following syntax, using e for *core expressions*, and s for XPath *steps*:

$$e ::= \textsf{for } \$x \textsf{ in } e_1 \textsf{ return } e_2 \mid \textsf{order } \$x \textsf{ in } e_1 \textsf{ by } e_2$$
$$\mid \textsf{let } \$x := e_1 \textsf{ return } e_2 \mid \$x \mid \textsf{if } (e_1) \textsf{ then } e_2 \textsf{ else } e_3$$

$$\mid \ell \mid () \mid e_1{,}e_2 \mid \textbf{element}\ q\ \{e\} \mid \$x/s \mid f(e_1,\ldots,e_n)$$
$$\mid [a_1 : e_1;\ldots;a_n : e_n] \mid \$x.a \mid e_1 +\!\!+ e_2 \mid \textbf{do insert}\ e_1\ \textbf{into}\ e_2 \mid \textbf{do delete}\ e$$

$$s ::= \textbf{child}::q \mid \textbf{descendant}::q \mid \ldots$$

$\$x$ (and other $-names) denotes *variables*, q denotes XML "qualified" *element names*, f denotes *function names*, a denotes *field names*, and ℓ denotes literals including numbers and strings; finally we have left the exact list of steps unspecified as our analysis does not depend on the specific steps. We allow the usual XPath/XQuery shorthands, in particular the **child**:: axis can be omitted, // abbreviates the **descendant**:: axis, and we will write certain built-in functions with traditional infix notation (such as $e_1 = e_2$ for equal(e_1, e_2)).

Our core differs from the W3C core [7] in three important ways: (1) it adds side-effects, in the form of two operations to update XML data in-place, which are executed immediately (2) it adds an explicit **order by** expression for sorting (a construction the W3C semantics does not specify), and (3) it adds *tuples* (*a.k.a. records*) which are finite mappings of field names to values: $[a_1 : e_1;\ldots;a_n : e_n]$ constructs a new tuple that maps each distinct field name a_i to the value of the corresponding e_i, $\$x.a$, extracts the value of the a field from the tuple value of $\$x$, and $e_1 +\!\!+ e_2$, constructs a new tuple with the combined fields from two existing tuples.

Since we use the core also as a target language for the algebra's semantics, we adopt record subtyping, *i.e.*, every tuple type specifies some fields that are guaranteed present, but more fields may be found in the typed value. This is notoriously incompatible with record concatenation: from $e_1 : [a : t]$ and $e_2 : [\ ]$ one cannot deduce that $e_1 +\!\!+ e_2 : [a : t]$, since e_2 may actually include an a field with an incompatible type. To solve the problem, we first adopt the following definition for the semantics of tuple concatenation for the case when the same field appears in both v_1 and v_2.

1. if $v_1.a = xv_1$ and $v_2.a = xv_2$ and xv_1 is different from xv_2, then $v_1 +\!\!+ v_2$ raises an error (informally "concatenation failure,");
2. if $v_1.a = xv = v_2.a$, then $v_1 +\!\!+ v_2$ associates a with xv.

We prove below that the translation of XQuery only generates well-typed *linear* core expressions (to be defined later), which never raise a concatenation failure. However, case (2) above must be allowed since it actually happens in the linear expressions that derive from XQuery translation.

Definition 2.2 (core semantics). The dynamic semantics of the core is defined by the judgment

$$\Sigma; \sigma \vdash e \Rightarrow v'; \sigma'$$

Σ is the dynamic environment mapping free variables of e to values (defined below). σ is a store mapping XML nodes *id*s to their value, and is used to support features such as node creation, node identity, backward navigation, and tree update. e is a core expression, v' the computed value, and σ' the resulting

store. (The actual definition is standard, apart from tuple concatenation which we commented on above, and can be found in the Appendix.)

Values are partitioned into two classes, XML values xv and table values tv. XML values are sequences of XML items iv, while table values are sequences of tuples. In both cases we identify a single item, or tuple, with the corresponding sequence of one element.

Definition 2.3 (values). The values, relative to a store σ, are given by

$$\begin{aligned} v &::= xv \mid tv && \text{(value)} \\ xv &::= xv_1, xv_2 \mid () \mid iv && \text{(XML value)} \\ tv &::= tv_1, tv_2 \mid () \mid [a_1 : xv_1; \ldots; a_n : xv_n] && \text{(table value)} \\ iv &::= \ell \mid id \qquad \text{with } id \in \sigma && \text{(item value)} \end{aligned}$$

where ℓ denotes literals. The fields of a tuple value are distinct and unordered.

The type system for the core should play two roles: (a) checking that predefined functions and operators are applied to arguments of the correct type, as it happens with the XQuery type system; (b) checking that tuple deconstruction and concatenation are correctly applied. Such a type system can be defined by enriching the XQuery type system with tuple types. To simplify the presentation, we follow here a much leaner approach, where all the types for the instances of the XQuery Data Model [10] are merged into *Item*, i.e. we focus on the (b) role only, since nothing is new, with respect to XQuery, on the (a) role. Similarly to values, types are partitioned into XML types xt and table types tt.

Definition 2.4 (types).

$$\begin{aligned} t &::= xt \mid tt && \text{(type)} \\ xt &::= \{xt\} \mid \mathit{Item} && \text{(XML type)} \\ tt &::= \{tt\} \mid [r] && \text{(table type)} \\ r &::= a : xt \mid r_1; r_2 \mid \epsilon && \text{(fields)} \\ ft &::= (t_1, \ldots, t_n) \to t && \text{(function type)} \end{aligned}$$

Tuple types are understood as follows:

- $[\epsilon]$ is the type of tuples with no fields, which we write $[]$.
- $[a : xt]$ is the type of tuples that map the field a to an XML value of type xt, and may be either defined or undefined on the other fields.
- $[r_1; r_2]$ is undefined if r_1 and r_2 map the same field name to two different types; otherwise, it is the intersection of types $[r_1]$ and $[r_2]$.

Hence, record fields are subject to the equalities $(r_1; r_2); r_3 = r_1; (r_2; r_3)$, $r_1; r_2 = r_2; r_1$, $a : xt; a : xt = a : xt$, and $r; \epsilon = r$, where $r_1 = r_2$ means that we identify them in every context (type equality, type rules, type semantics). Finally, since value sequences are flat, for any type t, $\{\{t\}\} = \{t\}$.

Definition 2.5 (core type semantics). The *semantics* of a type in a store, $\mathcal{T}[\![t]\!]_\sigma$, is defined as follows:

- $\mathcal{T}[\![Item]\!]_\sigma$ contains all node ids that are bound in σ, and all literal values;
- $\mathcal{T}[\![\{t\}]\!]_\sigma$ is the set of all finite sequences of elements of $\mathcal{T}[\![t]\!]_\sigma$; a single element of $\mathcal{T}[\![t]\!]_\sigma$ belongs to this set, and is equivalent to a singleton sequence;
- $\mathcal{T}[\![[a_1 : t_1; \ldots; a_n : t_n]]\!]_\sigma$ is the set of all functions that, for i in $1 \ldots n$, map a_i to an element of $\mathcal{T}[\![t_i]\!]_\sigma$; an element of $\mathcal{T}[\![[a_1 : t_1; \ldots; a_n : t_n]]\!]_\sigma$ may also be defined on any field name that is not specified in the type.
- $\mathcal{T}[\![(t_1, \ldots, t_n) \to t]\!]_\sigma$ is the set of all functions that, applied to n arguments in $\mathcal{T}[\![t_1]\!]_\sigma \ldots \mathcal{T}[\![t_n]\!]_\sigma$, return a value in $\mathcal{T}[\![t]\!]_\sigma$.

The fact that a tuple type does not give information about the fields that are not explicitly specified, and the fact that a single element is identified with a singleton sequence, lead to the following subtyping relation.

Definition 2.6 (subtyping). The *subtyping* relation $\leq:$ is the transitive homomorphic closure over types of the relation defined by

$$[r_1; r_2] \leq: [r_1] \qquad\qquad t \leq: \{t\}$$

Definition 2.7 (core typing). The judgment $\Gamma \vdash e : t$ holds iff it can be proved by the rules of Fig. 1, where:

1. Γ is a *type environment* which associates each free variable $\$x$ of e with a type $\Gamma(\$x)$, and each predefined function f with its function type $\Gamma(f)$.
2. $(\Gamma, \$x \mapsto t)$ denotes a new type environment where $\$x$ is assigned the type t instead of what it was in Γ; the empty type environment is written $()$.

We generalize the notions of typing to whole type environments: $\Sigma : \Gamma$ means that every variable bound by Σ is typed as specified by Γ, and $\mathcal{T}[\![\$x_1 : t_1; \ldots; \$x_n : t_n]\!]_\sigma$ is the set of all functions that, for i in $1 \ldots n$, map $\$x_i$ to an element of $\mathcal{T}[\![t_i]\!]_\sigma$.

Unfortunately, this type-system is not sound in general; for example, record concatenation fails in the two well-typed expressions below:

(1) [a:1] ++ [a:2]
(2) **let** $\$x$:= (**for** $\$w$ **in** (1,2) **return** [a:$\$w$]) **return**
for $\$y_1$ **in** $\$x$ **return for** $\$y_2$ **in** $\$x$ **return** ($\$y_1$ ++ $\$y_2$)

However, the type-system *is* sound when we restrict the attention to a *linear* subset of the language. Informally, a closed core expression is linear if none of the following non-linearity conditions apply.

1. *double construction*: the presence of two distinct constructors for a field a is "non-linear" (case (1) above); double construction is only allowed in independent subexpressions, as in **if** ([a:1]) **then** [a:2] **else** [a:3].
2. *non-linear let-variables*: two distinct uses of a **let** variable are "non-linear" (like the occurrences of $\$x$ in case (2) above); as in the *double construction* case, double use is allowed in independent code branches.

$$\frac{\Gamma \vdash e : t_1 \quad t_1 \leq: t_2}{\Gamma \vdash e : t_2}\ (\text{sub}) \qquad \frac{\Gamma(\$x) = t}{\Gamma \vdash \$x : t}\ (\text{var}) \qquad \frac{\Gamma \vdash e_1 : \{t_1\} \quad \Gamma \vdash e_2 : t_2 \quad \Gamma \vdash e_3 : t_2}{\Gamma \vdash \textsf{if}\ (e_1)\ \textsf{then}\ e_2\ \textsf{else}\ e_3 : t_2}\ (\text{if})$$

$$\frac{\Gamma \vdash e_1 : \{t_1\} \quad (\Gamma, \$x \Rightarrow t_1) \vdash e_2 : t_2}{\Gamma \vdash \textsf{for}\ \$x\ \textsf{in}\ e_1\ \textsf{return}\ e_2 : \{t_2\}}\ (\text{for}) \qquad \frac{\Gamma \vdash e_1 : \{t_1\} \quad (\Gamma, \$x \Rightarrow t_1) \vdash e_2 : \{\mathit{Item}\}}{\Gamma \vdash \textsf{order}\ \$x\ \textsf{in}\ e_1\ \textsf{by}\ e_2 : \{t_1\}}\ (\text{order})$$

$$\frac{\Gamma \vdash e_1 : t_1 \quad (\Gamma, \$x \Rightarrow t_1) \vdash e_2 : t_2}{\Gamma \vdash \textsf{let}\ \$x := e_1\ \textsf{return}\ e_2 : t_2}\ (\text{let}) \qquad \frac{}{\Gamma \vdash \ell : \mathit{Item}}\ (\text{literal}) \qquad \frac{}{\Gamma \vdash () : \{t\}}\ (\text{empty})$$

$$\frac{\Gamma \vdash e_1 : \{t\} \quad \Gamma \vdash e_2 : \{t\}}{\Gamma \vdash e_1, e_2 : \{t\}}\ (\text{comma}) \qquad \frac{\Gamma \vdash e : \{\mathit{Item}\}}{\Gamma \vdash \textsf{element}\ q\ \{e\} : \mathit{Item}}\ (\text{element}) \qquad \frac{\Gamma \vdash \$x : \mathit{Item}}{\Gamma \vdash \$x/s : \{\mathit{Item}\}}\ (\text{step})$$

$$\frac{\Gamma(f) = (\{xt_1\}, \ldots, \{xt_n\}) \to xt \quad \forall i \in 1..n\colon \Gamma \vdash e_i : \{xt_i\}}{\Gamma \vdash f(e_1, \ldots, e_n) : xt}\ (\text{fun}) \qquad \frac{\Gamma \vdash \$x : [a : xt]}{\Gamma \vdash \$x.a : xt}\ (\text{field})$$

$$\frac{\forall i \in 1..n\colon \Gamma \vdash e_i : xt_i}{\Gamma \vdash [a_1 : e_1; \ldots; a_n : e_n] : [a_1 : xt_1; \ldots; a_n : xt_n]}\ (\text{tuple}) \qquad \frac{\Gamma \vdash e_1 : [r_1] \quad \Gamma \vdash e_2 : [r_2]}{\Gamma \vdash e_1 +\!\!+ e_2 : [r_1; r_2]}\ (\text{concat})$$

$$\frac{\Gamma \vdash e : \{\mathit{Item}\}}{\Gamma \vdash \textsf{do delete}\ e : \{\mathit{Item}\}}\ (\text{delete}) \qquad \frac{\Gamma \vdash e_1 : \{\mathit{Item}\} \quad \Gamma \vdash e_2 : \{\mathit{Item}\}}{\Gamma \vdash \textsf{do insert}\ e_1\ \textsf{into}\ e_2 : \{\mathit{Item}\}}\ (\text{insert})$$

Fig. 1. Type rules for the core.

The above definition is extended to pairs (Σ, e) formed by an expression and a dynamic environment that defines the free variables of e, so that the evaluation of a linear expression only involves linear (Σ, e) pairs. With these tools, we can prove the following theorem, which will allow us to prove that the semantics of any well-typed XQueryU or algebraic expression is always well-defined, despite the combined use of subtyping and record concatenation in the core.

Theorem 2.8 (soundness of typing). *For any expression e linear for $\Sigma : \Gamma$, type t such that $\Gamma \vdash e : t$, and store σ such that $\Sigma \in \mathcal{T}[\![\Gamma]\!]_\sigma$, there exist $v'; \sigma'$ such that $\Sigma; \sigma \vdash e \Rightarrow v'; \sigma'$ and $v' \in \mathcal{T}[\![t]\!]_{\sigma'}$.*

3 XQuery

We define here XQueryU, a minimal subset of XQuery with immediate updates.

Definition 3.1 (XQueryU syntax). XQueryU syntax consists of *expressions* E, where F denotes FLWOR expressions, and s denotes XPath steps.

$$E ::= F \mid () \mid E_1\texttt{,}E_2 \mid \$x \mid \texttt{if (}E_1\texttt{) then }E_2\texttt{ else }E_3 \mid f\texttt{(}E_1\texttt{,}\ldots\texttt{,}E_n\texttt{)}$$
$$\mid \texttt{element }q\texttt{ \{}E\texttt{\}} \mid E/s \mid \ell \mid \texttt{do insert }E_1\texttt{ into }E_2 \mid \texttt{do delete }E$$

$$F ::= \texttt{for }\$x\texttt{ in }E\ F \mid \texttt{let }\$x\texttt{ := }E\ F \mid \texttt{where }E\ F \mid \texttt{order by }E\ F \mid \texttt{return }E$$

with the standard constraints that each FLWOR-expression must have at least one `for` or `let` clause, `where` clauses cannot be followed by `for` or `let` clauses, and `order by` clauses can only be followed by a `return` clause.

Definition 3.2 (XQuery semantics). The semantics of an XQueryU expression E is given by the translation $\mathcal{X}[\![E]\!]^{\varrho}_{\$t}$ defined in the following table, provided $\$t$ is a core variable of tuple type and ϱ is a mapping from XQueryU variables to core field names, which must map all free variables of E to fields defined by $\$t$.

Most of the XQueryU operators are mapped to the core by homomorphism. We use **do insert** as an example, and give the non-homomorphic cases:

$$\mathcal{X}[\![\texttt{do insert}\ E_1\ \texttt{into}\ E_2]\!]^{\varrho}_{\$t} = \textbf{do insert}\ \mathcal{X}[\![E_1]\!]^{\varrho}_{\$t}\ \textbf{into}\ \mathcal{X}[\![E_2]\!]^{\varrho}_{\$t}$$

$$\mathcal{X}[\![\$x]\!]^{\varrho}_{\$t} = \$t.a \qquad \text{where } a = \varrho(\$x)$$

$$\mathcal{X}[\![E/s]\!]^{\varrho}_{\$t} = \textbf{for}\ \$dot\ \textbf{in}\ (\mathcal{X}[\![E]\!]^{\varrho}_{\$t})\ \textbf{return}\ \$dot/s$$

$$\mathcal{X}[\![F]\!]^{\varrho}_{\$t} = \mathcal{X}^*[\![F]\!]^{\varrho}_{\$t} \qquad \text{where } \mathcal{X}^* \text{ is defined below}$$

$$\mathcal{X}^*[\![\texttt{for}\ \$x\ \texttt{in}\ E\ F]\!]^{\varrho}_{e} = \mathcal{X}^*[\![F]\!]^{(\varrho,\$x\mapsto a)}_{e_1} \qquad \text{where } a \text{ is a fresh field name, and}$$
$$e_1 = \textbf{for}\ \$t\ \textbf{in}\ e\ \textbf{return for}\ \$v\ \textbf{in}\ (\mathcal{X}[\![E]\!]^{\varrho}_{\$t})\ \textbf{return}\ \$t{+\!\!+}\,[a\colon \$v]$$

$$\mathcal{X}^*[\![\texttt{let}\ \$x\ \texttt{:=}\ E\ F]\!]^{\varrho}_{e} = \mathcal{X}^*[\![F]\!]^{(\varrho,\$x\mapsto a)}_{e_1} \qquad \text{where } a \text{ is a fresh field name, and}$$
$$e_1 = \textbf{for}\ \$t\ \textbf{in}\ e\ \textbf{return}\ \$t{+\!\!+}\,[a\colon \mathcal{X}[\![E]\!]^{\varrho}_{\$t}]$$

$$\mathcal{X}^*[\![\texttt{order by}\ E\ F]\!]^{\varrho}_{e} = \mathcal{X}^*[\![F]\!]^{\varrho}_{\textbf{order}\ \$t\ \textbf{in}\ e\ \textbf{by}\ (\mathcal{X}[\![E]\!]^{\varrho}_{\$t})}$$

$$\mathcal{X}^*[\![\texttt{where}\ E\ F]\!]^{\varrho}_{e} = \mathcal{X}^*[\![F]\!]^{\varrho}_{\textbf{for}\ \$t\ \textbf{in}\ e\ \textbf{return if}\ (\mathcal{X}[\![E]\!]^{\varrho}_{\$t})\ \textbf{then}\ \$t\ \textbf{else}\ ()}$$

$$\mathcal{X}^*[\![\texttt{return}\ E]\!]^{\varrho}_{e} = \textbf{for}\ \$t\ \textbf{in}\ e\ \textbf{return}\ (\mathcal{X}[\![E]\!]^{\varrho}_{\$t})$$

The most notable aspect of those rules is how the result of compilation for prior clauses in a FLWOR is passed as a parameter (as subscript) to the auxiliary $\mathcal{X}^*$ translation judgment.

$\mathcal{X}[\![E]\!]^{\varrho}_{\$t}$ is always well-defined, but it is only guaranteed to be well-typed in a specific static context, specified by Theorem 3.3 below. Informally, $\$t$ collects values for the free variables of E which is why ϱ must map these variables to field names of $\$t$. Similarly, in the helper translation $\mathcal{X}^*[\![F]\!]^{\varrho}_{e}$, e is an expression producing the tuple stream used to evaluate F, and ϱ maps the free variables of F to the field names of this tuple stream. To formalize these requirements we equip XQueryU with a type judgement, $\Gamma \vdash E : xt$, similar to the core typing of Def. 2.7, which we do not specify here for space reasons. We map type environments to tuple types by defining $\varrho(\$x_1{:}xt_1; \ldots; \$x_n{:}xt_n) = [\varrho(\$x_1){:}\{xt_1\}; \ldots; \varrho(\$x_n){:}\{xt_n\}]$, where each xt_i is mapped to $\{xt_i\}$ (*i.e.*, to $\{Item\}$) because we map `let` into **for**, hence the core type system sometimes infers sequence types in places where XQueryU typing was more precise. This is also reflected in the statement of the type preservation theorem. It would be easy to have exact type preservation, using a finer type system for the core, but this paper is focused on the use of tuple types to map static environments, and the other typing aspects are kept minimal by design.

We can finally show that the translation of well-typed terms is well-typed and linear, hence, thanks to Theorem 2.8, it is always well defined.

Theorem 3.3 (type preservation). *If $\varrho(\Gamma)$ is well defined, then:*

$$\Gamma \vdash E : xt \qquad \Rightarrow (\$t : \varrho(\Gamma)) \vdash \mathcal{X}[\![E]\!]^{\varrho}_{\$t} : \{xt\}$$
$$(\$t : tt) \vdash e : \{\varrho(\Gamma)\} \wedge \Gamma \vdash F : xt \ \Rightarrow (\$t : tt) \vdash \mathcal{X}^*[\![F]\!]^{\varrho}_{e} : \{xt\}$$

Theorem 3.4 (linearity). *For any E, ϱ, tv, where ϱ is defined on all free variables of E, and $tv = [\varrho(\$x_1) : xv_1; \ldots; \varrho(\$x_n) : xv_n]$ is defined on the whole image of ϱ, the term $\mathcal{X}[\![E]\!]^{\varrho}_{\$t}$ is linear with $(\$t \mapsto tv)$.*

Corollary 3.5 (XQueryU semantics). *For any $\Gamma, E, xt, \varrho, \sigma, tv$, if $\Gamma \vdash E : xt$, $\varrho(\Gamma)$ is defined, $tv = [\varrho(\$x_1) : xv_1; \ldots; \varrho(\$x_n) : xv_n]$ with $xv_i \in \mathcal{T}[\![\Gamma(x_i)]\!]_\sigma$, then there exist $v'; \sigma'$ such that $(\$t \mapsto tv); \sigma \vdash \mathcal{X}[\![E]\!]^{\varrho}_{\$t} \Rightarrow v'; \sigma'$ and $v' \in \mathcal{T}[\![t]\!]_{\sigma'}$.*

4 Algebra

We formally specify the semantics and type system for an existing nested-relational algebra for XQuery [18, 21]. Most other database algebras are quite similar, so most of the treatment proposed here should apply quite directly to other relational or nested-relational algebras. From a functional programming perspective, database algebras are first order languages, where efforts are taken in order to avoid any manipulation of functions. For example, the projection operator from relational algebra, usually written $\pi_\phi R$, is similar to a map and applies ϕ to every element of R. However, traditional database algebras usually do not allow ϕ to be a function from tuples to tuples, but always use a variable-free syntax [20], which significantly simplifies the analysis and rewriting of algebraic terms. Algebras for object databases sometimes depart from this, since methods have to be formalized, and we have seen a drift towards higher-order algebras, where functions and lambda-binders play a role. XQuery seems to call for that approach, since the language is functional, and also because every expression is always evaluated with respect to an implicit context item, which means that every expressions denotes a function.

We propose a different approach that merges the advantages of binder-free syntax and the expressivity of higher-order. In our algebra, every term (or *plan*) denotes a first-order function, that yields a result when applied to a context tuple. Every n-ary algebra operator (with $n > 0$), such as Map or Select, denotes a second-order function, which yields a first-order plan when applied to first-order subplans. We will show how this approach gives all the expressive power we need, while avoiding the need for higher-order syntax and rewriting.

Example 4.1. Consider the following XQuery expression:

```
for $x in $doc//a where $x/empno≥1 return $x
```

Database compilers compile this into a *query plan* similar to the following, denoting a function to be applied to a context tuple where the #doc field is defined.

```
Map{#x}(Select{TreeJoin[empno](#x) ≥1}
                (MapConcat{Map{[x:ID]}(TreeJoin[//a](#doc))}
                          (ID)))
```

To illustrate the first order nature of each operator, the same plan with explicit binders would look as follows.

$$\frac{}{\mathsf{ID} : t \to t}\ (\text{ID}) \qquad \frac{}{\ell : t \to \mathit{Item}}\ (\text{Literal}) \qquad \frac{p : t \to xt}{[a : p] : t \to [a : xt]}\ (\text{Tuple}) \qquad \frac{}{\#a : [a : xt] \to xt}\ (\text{Field})$$

$$\frac{p_2 : t \to \{[r]\} \quad p_1 : [r] \to \{\mathit{Item}\}}{\mathsf{Select}\{p_1\}(p_2) : t \to \{[r]\}}\ (\text{Select}) \qquad \frac{p_2 : t \to \{[r]\} \quad p_1 : [r] \to \{\mathit{Item}\}}{\mathsf{OrderBy}\{p_1\}(p_2) : t \to \{[r]\}}\ (\text{OrderBy})$$

$$\frac{p_2 : t \to \{[r_2]\} \quad p_1 : [r_2] \to \{[r_1]\}}{\mathsf{MapConcat}\{p_1\}(p_2) : t \to \{[r_1; r_2]\}}\ (\text{MapConcat}) \qquad \frac{p_2 : t \to \{t'\} \quad p_1 : t' \to t''}{\mathsf{Map}\{p_1\}(p_2) : t \to \{t''\}}\ (\text{Map})$$

$$\frac{}{\mathsf{Empty}() : t \to \{\mathit{Item}\}}\ (\text{Empty}) \qquad \frac{p_1 : t \to \{\mathit{Item}\} \quad p_2 : t \to \{\mathit{Item}\}}{\mathsf{Sequence}(p_1, p_2) : t \to \{\mathit{Item}\}}\ (\text{Seq})$$

$$\frac{p : t \to \{\mathit{Item}\}}{\mathsf{TreeJoin}[s](p) : t \to \{\mathit{Item}\}}\ (\text{TreeJoin}) \qquad \frac{p_1 : t \to \{\mathit{Item}\} \quad p_2 : t \to u \quad p_3 : t \to u}{\mathsf{Conditional}(p_1, p_2, p_3) : t \to u}\ (\text{If})$$

$$\frac{\Gamma(f) = (t'_1, \ldots, t'_n) \to t' \quad p_1 : t \to t'_1 \quad \ldots \quad p_n : t \to t'_n}{\mathsf{Call}[f](p_1, \ldots, p_n) : t \to t'}\ (\text{Call})$$

$$\frac{p_1 : t \to \{\mathit{Item}\} \quad p_2 : t \to \{\mathit{Item}\}}{\mathsf{Insert}(p_1, p_2) : t \to \{\mathit{Item}\}}\ (\text{Insert}) \qquad \frac{p : t \to \{\mathit{Item}\}}{\mathsf{Delete}(p) : t \to \{\mathit{Item}\}}\ (\text{Delete})$$

$$\frac{p : t \to u \quad p : t^- \leq: t \quad p : u \leq: u^+}{p : t^- \to u^+}\ (\text{Sub})$$

Fig. 2. Type rules for the base algebra.

```
λt0→ Map (λt1→ t1.x)
         (Select (λt2→ t2.x/empno ≥1)
                 (MapConcat (λt3→ Map (λt4→ [x : t4]) (t3.doc//a))
                            (t0)))
```

Due to lack of space, we focus on a base algebra which contains only the algebraic operators that are needed for compilation, and we ignore the additional operators needed for optimization purposes.

Definition 4.2 (base algebra syntax). For unoptimized *query plans* we use the following basic syntax:

$$\begin{aligned} p ::= {} & \mathsf{ID} \mid \mathsf{Empty}() \mid \mathsf{Sequence}(p_1, p_2) \mid \ell \mid \mathsf{Element}[q](p) \\ & \mid \mathsf{Select}\{p_1\}(p_2) \mid \mathsf{OrderBy}\{p_1\}(p_2) \mid \mathsf{Map}\{p_1\}(p_2) \mid \mathsf{MapConcat}\{p_1\}(p_2) \\ & \mid \mathsf{TreeJoin}[s](p) \mid \mathsf{Conditional}(p_1, p_2, p_3) \mid \mathsf{Call}[f](p_1, \ldots, p_n) \\ & \mid [a{:}p] \mid \#a \mid \mathsf{Insert}(p_1, p_2) \mid \mathsf{Delete}(p) \end{aligned}$$

The semantics of the basic query plans is given in the following table, through a translation to the core. Every plan p denotes a core expression $\mathcal{A}[\![p]\!]_{\$t}$ with one free variable $\$t$ for the input tuple stream. Every algebraic plan receives an input value, and, with the only exception of the leaf operators $\#a$, ID, ℓ and Empty, does not operate on it, but passes it to the subplans enclosed in round

brackets. Select, OrderBy, MapConcat and Map also apply their curly-brackets subplan to each value in the list returned by the round-brackets. Finally, the values returned by the subplans are acted upon.

$\mathcal{A}[\![\mathsf{ID}]\!]_{\$t} = \$t$
$\mathcal{A}[\![\mathsf{Sequence}(p_1, p_2)]\!]_{\$t} = (\mathcal{A}[\![p_1]\!]_{\$t}, \mathcal{A}[\![p_2]\!]_{\$t})$
$\mathcal{A}[\![\mathsf{Empty}()]\!]_{\$t} = ()$
$\mathcal{A}[\![\mathsf{Scalar}[\ell]()]\!]_{\$t} = \ell$
$\mathcal{A}[\![\mathsf{Element}[q](p)]\!]_{\$t} = \textbf{element } q \ \{\mathcal{A}[\![p]\!]_{\$t}\}$
$\mathcal{A}[\![\mathsf{Select}\{p_1\}(p_2)]\!]_{\$t} = \textbf{for } \$t_1 \textbf{ in } \mathcal{A}[\![p_2]\!]_{\$t} \textbf{return if } (\mathcal{A}[\![p_1]\!]_{\$t_1}) \textbf{ then } \$t_1 \textbf{ else } ()$
$\mathcal{A}[\![\mathsf{TreeJoin}[s](p)]\!]_{\$t} = \textbf{for } \$t_1 \textbf{ in } \mathcal{A}[\![p]\!]_{\$t} \textbf{return } \$t_1/s$
$\mathcal{A}[\![\mathsf{Map}\{p_1\}(p_2)]\!]_{\$t} = \textbf{for } \$t_1 \textbf{ in } \mathcal{A}[\![p_2]\!]_{\$t} \textbf{return} \mathcal{A}[\![p_1]\!]_{\$t_1}$
$\mathcal{A}[\![\mathsf{MapConcat}\{p_1\}(p_2)]\!]_{\$t} = \textbf{for } \$t_1 \textbf{ in } \mathcal{A}[\![p_2]\!]_{\$t} \textbf{return}$
$\qquad\qquad\qquad\qquad\qquad\qquad \textbf{for } \$t_2 \textbf{ in } \mathcal{A}[\![p_1]\!]_{\$t_1} \textbf{return } \$t_1 +\!\!+ \$t_2$
$\mathcal{A}[\![\mathsf{OrderBy}\{p_1\}(p_2)]\!]_{\$t} = \textbf{order } \$t_1 \textbf{ in } \mathcal{A}[\![p_2]\!]_{\$t} \textbf{by} \mathcal{A}[\![p_1]\!]_{\$t_1}$
$\mathcal{A}[\![\mathsf{Conditional}(p_1, p_2, p_3)]\!]_{\$t} = \textbf{if } (\mathcal{A}[\![p_1]\!]_{\$t}) \textbf{ then } \mathcal{A}[\![p_2]\!]_{\$t} \textbf{else } \mathcal{A}[\![p_3]\!]_{\$t}$
$\mathcal{A}[\![\mathsf{Call}[f](p_1, \ldots, p_n)]\!]_{\$t} = f(\mathcal{A}[\![p_1]\!]_{\$t}, \ldots, \mathcal{A}[\![p_n]\!]_{\$t})$
$\mathcal{A}[\![[a : p]]\!]_{\$t} = [a : \mathcal{A}[\![p]\!]_{\$t}]$
$\mathcal{A}[\![\#a]\!]_{\$t} = \$t.a$
$\mathcal{A}[\![\mathsf{Insert}(p_1, p_2)]\!]_{\$t} = \textbf{do insert} \mathcal{A}[\![p_1]\!]_{\$t} \textbf{ into } \mathcal{A}[\![p_2]\!]_{\$t}$
$\mathcal{A}[\![\mathsf{Delete}(p)]\!]_{\$t} = \textbf{do delete } \mathcal{A}[\![p]\!]_{\$t}$

Definition 4.3 (application of a plan). The notation $\mathcal{A}[\![p]\!]_v^\sigma$ denotes the value-store pair $v'; \sigma'$ such $\$t \mapsto v; \sigma \vdash \mathcal{A}[\![p]\!]_{\$t} \Rightarrow v'; \sigma'$. We will also use the same notation to denote just the value component v', when this is clear from the context.

Every plan has a first order type $t \to t'$, where t and t' are defined exactly as in Definition 2.4; the two type languages coincide since the semantics of the algebra is given through a translation to the core.

Definition 4.4 (algebra typing). The algebra type system is defined by the rules in Figure 2; subtyping is defined as in the core.

Notice that Γ is treated as a constant here (with just the built-in function signatures) hence is not propagated within the rules. Tuple concatenation plays a key role here as in the core (whereas the distinction between $\{t\}$ and t is only relevant for optimizations purposes that we do not consider here). Once more, we are able to combine record concatenation with record subtyping by adopting a linearity constraint. We say that the query plan p *builds* a field a if it contains a tuple constructor for a, *i.e.*, a subplan $[a : p']$. Linearity then means that, informally, every tuple field is only built in one specific point of the code.

Definition 4.5 (linearity). A plan p is *linear* if no field is built by two distinct tuple constructors inside p.

A plan p is *linear with a tuple type* $[a_1 : t_1; ...; a_n : t_n]$ or with a tuple sequence type $\{[a_1 : t_1; ...; a_n : t_n]\}$, if it is linear and no field among $a_1, \ldots, a_n$ is built by p. A piece of code p is linear with an XML type iff p is linear.

Definition 4.6 ($\pi_t(v)$). The projection $\pi_t(v)$ of a value v over a type t such that $v \in \mathcal{T}[\![t]\!]_\sigma$ is defined by cases on t, as follows. $\pi_{[a_1:t_1,\ldots,a_n:t_n]}(v)$ is a tuple that coincides with v on the fields $a_1, \ldots, a_n$, and is undefined on the others. $\pi_{\{tt\}}(v_1, \ldots, v_n)$ is equal to $\pi_{\{tt\}}(v_1), \ldots, \pi_{\{tt\}}(v_n)$. Finally, $\pi_{xt}(v) = v$.

We can now state the type soundness of the algebra. The theorem is not trivial, because it requires that the extra fields in the input tuple are removed by projection ($\pi_t(tv)$), but the thesis $v' \in \mathcal{T}[\![t']\!]_{\sigma'}$ does not exclude the presence of extra fields in the result. This means that the theorem cannot be proved directly by induction, but we need to resort on linearity to prove that the extra fields are actually harmless. In a sense, this theorem specifies that any optimizer that preserves types and linearity never needs to insert extra projections.

Theorem 4.7 (type soundness). *If $p : t \to t'$ and p is linear with t, then, for any σ and any tuple $tv \in \mathcal{T}[\![t]\!]_\sigma$, then $\mathcal{A}[\![p]\!]^\sigma_{\pi_t(tv)}$ is well defined, and, if $(v', \sigma') = \mathcal{A}[\![p]\!]^\sigma_{\pi_t(tv)}$, then $v' \in \mathcal{T}[\![t']\!]_{\sigma'}$.*

4.1 Compilation

We give now a formal definition of the algebraic compilation $\mathcal{C}[\![E]\!]^\varrho$ of an XQueryU expression E, in the table below. ϱ is a mapping from free variables in E to core field names, and a is a fresh field name in cases *for* and *let*.

$\mathcal{C}[\![F]\!]^\varrho = \mathcal{C}^*[\![F]\!]^\varrho_{\mathsf{ID}}$ where $\mathcal{C}^*$ is defined below
$\mathcal{C}[\![\texttt{()}]\!]^\varrho = \mathsf{Empty()}$
$\mathcal{C}[\![E_1\texttt{,}E_2]\!]^\varrho = \mathsf{Sequence}(\mathcal{C}[\![E_1]\!]^\varrho, \mathcal{C}[\![E_2]\!]^\varrho)$
$\mathcal{C}[\![\$x]\!]^\varrho = \#a$ with $a = \varrho(\$x)$
$\mathcal{C}[\![\texttt{if (}E_1\texttt{) then }E_2\texttt{ else }E_3]\!]^\varrho = \mathsf{Conditional}(\mathcal{C}[\![E_1]\!]^\varrho, \mathcal{C}[\![E_2]\!]^\varrho, \mathcal{C}[\![E_3]\!]^\varrho)$
$\mathcal{C}[\![f\texttt{(}E_1, \ldots, E_n\texttt{)}]\!]^\varrho = \mathsf{Call}[f](\mathcal{C}[\![E_1]\!]^\varrho, \ldots, \mathcal{C}[\![E_n]\!]^\varrho)$
$\mathcal{C}[\![\texttt{element }q\texttt{ \{}E\texttt{\}}]\!]^\varrho = \mathsf{Element}[q](\mathcal{C}[\![E]\!]^\varrho)$
$\mathcal{C}[\![E\texttt{/}s]\!]^\varrho = \mathsf{TreeJoin}[s](\mathcal{C}[\![E]\!]^\varrho)$
$\mathcal{C}[\![\ell]\!]^\varrho_{\$t} = \ell$
$\mathcal{C}[\![\texttt{do insert }E_1\texttt{ into }E_2]\!]^\varrho = \mathsf{Insert}(\mathcal{C}[\![E_1]\!]^\varrho)\mathcal{C}[\![E_2]\!]^\varrho$
$\mathcal{C}[\![\texttt{do delete }E]\!]^\varrho = \mathsf{Delete}(\mathcal{C}[\![E]\!]^\varrho)$

$\mathcal{C}^*[\![\texttt{for }\$x\texttt{ in }E\ F]\!]^\varrho_p = \mathcal{C}^*[\![F]\!]^{(\varrho, \$x \mapsto a)}_{p_1}$ $p_1 = \mathsf{MapConcat}\{\mathsf{Map}\{[a : \mathsf{ID}]\}(\mathcal{C}[\![E]\!]^\varrho)\}(p)$
$\mathcal{C}^*[\![\texttt{order by }E\ F]\!]^\varrho_p = \mathcal{C}^*[\![F]\!]^\varrho_{p_1}$ $p_1 = \mathsf{OrderBy}\{\mathcal{C}[\![E]\!]^\varrho\}(p)$
$\mathcal{C}^*[\![\texttt{let }\$x\texttt{ := }E\ F]\!]^\varrho_p = \mathcal{C}^*[\![F]\!]^{(\varrho, \$x \mapsto a)}_{p_1}$ $p_1 = \mathsf{MapConcat}\{[a : \mathcal{C}[\![E]\!]^\varrho]\}(p)$
$\mathcal{C}^*[\![\texttt{where }E\ F]\!]^\varrho_p = \mathcal{C}^*[\![F]\!]^\varrho_{p_1}$ $p_1 = \mathsf{Select}\{\mathcal{C}[\![E]\!]^\varrho\}(p)$
$\mathcal{C}^*[\![\texttt{return }E]\!]^\varrho_p = \mathsf{Map}\{\mathcal{C}[\![E]\!]^\varrho\}(p)$

The compilation rules correspond strictly to those of [21] (except for some minor syntactic variations and the use of ϱ for explicit field naming). As in the rules that describe the semantics of XQueryU, the most interesting rules are those we have described with the helper function $\mathcal{C}^*[\![F]\!]^\varrho_p$, which compiles the "tail clauses" F of a FLWOR expression. Those rules are specified in the context of an operator p that generates the stream of tuples that are generated by the "head clauses" of the same FLWOR.

Example 4.8 (algebraic compilation). The example XQueryU from the "two semantics" part of the introduction compiles with the translation scheme to the algebraic expression in the "database algebra" part of the introduction. The following table shows how the application of $\mathcal{C}^*$ constructs the plan backwards:

i F_i	$p_i = \mathcal{C}^*[\![F_i]\!]_{\dots}^{\dots}$
1 `for $a in` ... F_2	$\mathsf{Map}\{[\mathsf{a{:}ID}]\}(\dots)$
2 `for $b in` ... F_3	$\mathsf{MapConcat}\{\mathsf{Map}\{[\mathsf{b{:}ID}]\}(\dots)\}(p_1)$
3 `where $a=$b` F_4	$\mathsf{Select}\{\#\mathsf{a}=\#\mathsf{b}\}(p_2)$
4 `return $a`	$\mathsf{Map}\{\#\mathsf{a}\}(p_3)$

The following theorem expresses the correctness of this compilation scheme, with respect to our semantics. Interestingly, the proof is done by merely applying a simple variant of the semantics provided in Section 4 to the result of compilation, and showing that it is *syntactically* equivalent to the semantics of the same query in the core. The variant is defined in the Appendix.

Theorem 4.9 (correctness). *For any environment Σ, stores σ, σ', XQueryU expression E, variable $t, field name assignment ϱ defined for all free variables in E, and value v',*

$$\Sigma, \sigma \vdash \mathcal{A}[\![\mathcal{C}[\![E]\!]^{\varrho}]\!]_{\$t} \Rightarrow v, \sigma' \quad \textit{iff} \quad \Sigma, \sigma \vdash \mathcal{X}[\![E]\!]^{\varrho}_{\$t} \Rightarrow v, \sigma'$$

5 Conclusion

In this paper, we showed how the nested-for and the tuple-stream semantics for FLWOR expressions diverge in presence of side-effects. We have formalized the compilation process for XQuery extended with side effects into a database algebra, and we have shown that this compilation scheme is sound for the tuple-stream semantics. This formalization shows that the tuple-stream semantics admits a simple implementation, based on normalization and list comprehensions, along the lines of the traditional implementation of main-memory programming language iterators. We are currently investigating optimization of database languages with side-effects based on that framework.

Acknowledgements. We would like to thank Limsoon Wong for clarifying some details about optimization in Kleisli, and Mary Fernández for feedback on earlier drafts of this paper.

References

1. M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. Gray, P. P. Griffiths, W. F. K. III, R. A. Lorie, P. R. McJones, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Wade, and V. Watson. System R: Relational approach to database management. *ACM Transactions on Database Systems*, 1(2):97–137, 1976.
2. S. Boag, D. Chamberlain, M. F. Fernández, D. Florescu, J. Robie, and J. Siméon. XQuery 1.0: An XML query language, W3C recommendation, 2007.

3. P. Buneman, L. Libkin, D. Suciu, V. Tannen, and L. Wong. Comprehension syntax. *SIGMOD Record*, 23(1):87–96, 1994.
4. D. Chamberlain, M. Carey, D. Florescu, D. Kossmann, and J. Robie. XQueryP: Programming with XQuery. In *XIME-P*, 2006.
5. D. Chamberlain, D. Florescu, and J. Robie. XQuery scripting extension 1.0 requirements, W3C working draft 23 march 2007. http://www.w3.org/TR/2007/WD-xquery-sx-10-requirements-20070323/, 2007.
6. D. Chamberlain, D. Florescu, and J. Robie. XQuery Update Facility, W3C working draft 11 july 2006, 2007.
7. 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 recommendation 24 january 2007, 2007.
8. L. Fegaras. Query unnesting in object-oriented databases. In *SIGMOD Conference*, pages 49–60, 1998.
9. L. Fegaras. Optimizing queries with object updates. *J. Intell. Inf. Syst.*, 12(2-3):219–242, 1999.
10. M. Fernández, A. Malhotra, J. Marsh, M. Nagy, and N. Walsh. XQuery 1.0 and XPath 2.0 data model (xdm). W3C Recommendation, Jan. 2007.
11. M. F. Fernández, J. Siméon, and P. Wadler. A semi-monad for semi-structured data. In *ICDT*, pages 263–300, 2001.
12. D. Florescu, A. Grünhagen, and D. Kossmann. XL: An XML programming language for Web service specification and composition. In *International conference on World Wide Web*, pages 65–76, May 2002.
13. G. Ghelli, C. Re, and J. Siméon. XQuery!: An XML query language with side effects. In *EDBT Workshops*, pages 178–191, 2006.
14. C. A. Gunter and J. C. Mitchell. *Theoretical Aspects of Object-Oriented Programming*. The MIT Press, 1994.
15. M. Harren, M. Raghavachari, O. Shmueli, M. G. Burke, R. Bordawekar, I. Pechtchanski, and V. Sarkar. Xj: facilitating xml processing in java. In *International conference on World Wide Web*, pages 278–287, 2005.
16. J. Hidders, J. Paredaens, and R. Vercammen. On the expressive power of xquery-based update languages. In *XSym*, pages 92–106, 2006.
17. The linq project. msdn.microsoft.com/XML/linqproject.
18. N. May, S. Helmer, and G. Moerkotte. Nested queries and quantifiers in an ordered context. In *ICDE*, pages 239–250, 2004.
19. G. Moerkotte. Building query compilers, draft manuscript. http://db.informatik.uni-mannheim.de/ moer, December 2005.
20. R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill, 2000.
21. C. Re, J. Siméon, and M. F. Fernández. A complete and efficient algebraic compiler for XQuery. In *ICDE*, page 14, 2006.
22. V. Tannen. Tutorial: Languages for collection types. In *PODS*, pages 150–154, 1994.
23. V. Tannen, P. Buneman, and L. Wong. Naturally embedded query languages. In *ICDT*, pages 140–154, 1992.
24. P. Trinder and P. Wadler. Improving list comprehension database queries. In *Fourth IEEE Region 10 Conference (TENCON)*, pages 186–192, Nov. 1989.
25. L. Wong. Kleisli, a functional query system. *Journal of Functional Programming*, 10(1):19–56, 2000.