On the denotational semantics of XML-Lambda[⋆]

Pavel Loupal[1] and Karel Richta[2]

[1] Department of Software Engineering, Czech Technical University, Faculty of Information Technology
Prague, Czech Republic
pavel.loupal@fit.cvut.cz

[2] Department of Software Engineering, Charles University, Faculty of Mathematics and Physics
Prague, Czech Republic
richta@ksi.mff.cuni.cz

Abstract. *The article deals with the denotational semantics of a special query language called XML-Lambda (abbreviated as XML-λ), which is based on the simply typed lambda calculus. The exact semantics allows experimentation with a language definition, prototyping of programs, and similar experiments. One of such experiment is the implementation of the XQuery language in the XML-λ environment. The main advantage of our approach is the possibility of a query optimizations in the XML-λ intermediate form. It is much more easier than optimizations based on the official W3C semantics. XML-λ is a part of more complex XML-λ Framework which serves for experimenting with the tools for XML processing.*

1 Introduction

In this paper, we define formally the semantics of XML-Lambda Query Language. From now on we will use abbreviation XML-λ. XML-λ employs the functional data model for XML data elaboration. The first idea for such an attitude was published in [5, 6]. This research brought in the key idea of a functional query processing with a wide potential that was later proven by a simple prototype implementation [7].

We can imagine two scenarios for this language; firstly, the language plays a role of a full-featured query language for XML (it has both formal syntax and semantics and there is also an existing prototype that acts as a proof-of-the-concept application). But there already exist standard approaches for XML querying – especially XQuery, with probably more appropriate syntax for users. So, there is no need to define any new query language.

In the second scenario, the XML-λ Query Language is utilized as an intermediate language for the description of XQuery semantics. In [4] we propose a novel method for XQuery evaluation based on the transformation of XQuery queries into their XML-λ equivalents and their subsequent evaluation. As an integral part of the work, we have designed and developed a prototype of an XML-λ query processor for validating the functional approach and experimenting with it. The main advantage of this concept is the possibility of a query optimizations in the XML-λ intermediate form. It is much more easier than optimizations when we use the official W3C semantics ([3]).

Since it is not possible to express the semantics of the whole XML-λ language in this contribution, the paper focuses chiefly on its main idea and concepts.

2 XML-λ Query Language

In this section, we briefly describe the XML-λ Query Language, a query language for XML based on the simply typed lambda calculus.

2.1 Language of terms

Typical query expression has a query part — an expression to be evaluated over data — and a constructor part that wraps a query result and forms the output. The XML-λ Query Language is based on λ-terms defined over the type system $\mathcal{T}_E$ as will be shown later. λ-calculus is a formal mathematical system for investigation of function definition and application. It was introduced by Alonzo Church and has been utilized in many ways. In this work, we use a variant of this formalism, the simply-typed λ-calculus, as a core for the XML-λ Query Language. We have gathered the knowledge from [8] and [1]. Our realization is enriched by usage of tuples.

The main constructs of the language are *variables*, *constants*, *tuples*, *projections*, and λ-calculus operations — *applications* and *abstractions*. The syntax is similar to λ-calculus expressions, thus the queries are structured as nested λ-expressions, i.e.:

$$\lambda \ldots (\lambda \ldots (expression) \ldots)$$

In addition, there are also typical constructs such as logical connectives, constants, comparison predicates, and a set of built-in functions.

[⋆] This work has been supported by the Ministry of Education, Youth and Sports under Research Program No. MSM 6840770014 and also by the grant project of the Czech Grant Agency No. GA201/09/0990.

Language of terms is defined inductively as the least set containing all terms created by the application of the following rules. Let $T, T_1, \ldots, T_n$, $n \geq 1$ be members of $\mathcal{T}_E$. Let $\mathcal{F}$ be a set of typed constants, and $\mathcal{V}$ an at most countable set of typed variables. Then:

1. *variable:* each variable (member of $\mathcal{V}$) of type T is a term of type T
2. *constant:* each constant (member of $\mathcal{F}$) of type T is a term of type T
3. *application:* if M is a term of type $((T_1,\ldots,T_n) \to T)$ and $N_1, \ldots, N_n$ are terms of the types $T_1, \ldots, T_n$, then $M(N_1, \ldots, N_n)$ is a term of the type T
4. λ-*abstraction:* if $x_1, \ldots, x_n$ are distinct variables of types $T_1, \ldots, T_n$ and M is a term of type T, then $\lambda x_1 : T_1, \ldots, x_n : T_1.(M)$ is a term of type $((T_1, \ldots, T_n) \to T)$
5. *n-tuple:* if $N_1,\ldots,N_n$ are terms of types $T_1,\ldots,T_n$, then $(N_1, \ldots, N_n)$ is a term of type $(T_1, \ldots, T_n)$
6. *projection:* if $(N_1, \ldots, N_n)$ is a term of type $(T_1, \ldots\ , T_n)$, then $N_1, \ldots, N_n$ are terms of types $T_1, \ldots, T_n$
7. *tagged term:* if N is a term of type $NAME$ and M is a term of type T then $N : M$ is a term of type $(\mathbf{E} \ \to T)$.

The set of abstract elements $\mathbf{E}$ serves as a notation for abstract positions in XML trees. Terms can be interpreted in a standard way by means of an interpretation assigning to each constant from $\mathcal{F}$ an object of the same type, and by a semantic mapping from the language of terms to all functions and Cartesian products given by the type system $\mathcal{T}_E$. Speaking briefly, an application is evaluated as an application of the associated function to given arguments, an abstraction 'constructs' a new function of the respective type. The tuple is a member of Cartesian product of sets of typed objects. A tagged term is interpreted as a function defined only for one $e \in \mathbf{E}$. It returns again a function.

3 Abstract syntax

As for evaluation of a query, we do not need its complete derivation tree; such information is too complex and superfluous. Therefore, in order to diminish the domain that needs to be described without any loss of precision, we employ the *abstract syntax.* With the abstract syntax, we break up the query into logical pieces that forming an abstract syntax tree carrying all original information constitute an internal representation suitable for query evaluation. We introduce syntactic domains for the language, i.e., logical blocks a query may consist of. Subsequently, we list all production rules. These definitions are later utilized in Section 4 within the denotational semantics.

3.1 Syntactic domains

By the term *syntactic domain*, we understand a logical part of a language. In Table 1, we list all syntactic domains of the XML-λ Query Language with their informal meaning. Notation Q : *Query* stands for the symbol Q representing a member of the *Query* domain.

Q : *Query*	XML-λ queries,
O : *Options*	XML input attachements,
C : *Constructor*	constructors of output results,
E : *Expression*	general expressions,
SQ : *SubQuery*	(nested) subqueries,
T : *Term*	sort of expression,
F : *Fragment*	sub-parts of a *Term*,
As : *Assignment*	variable assignments,
Flt : *Filter*	set pruning conditions,
FC : *FunctionCall*	either built-in or user-defined functions,
BinOp : *BinOperator*	binary logical operators,
RelOp : *RelOperator*	binary relational operators,
NF : *Nullary*	identifiers of nullary functions (subset of *Identifier*),
Proj : *Projection*	identifiers for projections (subset of *Identifier*),
B : *Boolean*	logical values,
N : *Numeral*	numbers,
D : *Digits*	digits,
S : *String*	character strings,
Id : *Identifier*	strings conforming to the *Name* syntactic rule in [2].

Table 1. Syntactic domains of the XML-λ Query Language.

3.2 Abstract production rules

The *abstract production rules* listed in Table 2 (written using EBNF) connect the terms of syntactic domains from the previous section into logical parts with suitable level of details for further processing. On the basis of these rules, we will construct the denotational semantics of the language.

4 Denotational semantics

We use *denotational semantics* for the description of the meaning of each XML-λ query. The approach is based on the idea that for each correct syntactic construct of the language we can define a respective meaning of it as a formal expression in another, well-known, notation. We can say that the program is the denotation of its meaning. The validity of the whole approach is based on structural induction; i.e, that the meaning of more complex expressions is defined on the basis

$Query$	$::=$ $Options\ Constructor\ Expression$
$Constructor$	$::=$ $ElemConstr+ \mid Identifier+$
$ElemConstr$	$::=$ $Name\ AttrConstr* \ (Identifier \mid ElemConstr)$
$AttrConstr$	$::=$ $Name\ Identifier$
$Expression$	$::=$ $Fragment$
$Fragment$	$::=$ $Nullary \mid Identifier \mid Term \mid Fragment\ Projection \mid SubQuery \mid FunctionCall \mid Numeral \mid String \mid Boolean$
$Term$	$::=$ $Boolean \mid Filter \mid$ 'not' $Term \mid Term\ BinOperator\ Term$
$Filter$	$::=$ $Fragment\ RelOperator\ Fragment$
$SubQuery$	$::=$ $Identifier+\ Expression$
$BinOperator$	$::=$ 'or' $\mid$ 'and'
$RelOperator$	$::=$ '<=' $\mid$ '<' $\mid$ '==' $\mid$ '!=' $\mid$ '>' $\mid$ '>='
$Numeral$	$::=$ Digit+ $\mid$ Numeral '.' Digit+
$Digit$	$::=$ '0'$\mid$'1'$\mid$'2'$\mid$'3'$\mid$'4'$\mid$'5'$\mid$'6'$\mid$'7'$\mid$'8'$\mid$'9'
$Identifier$	$::=$ $Name$
$Projection$	$::=$ $Identifier$

Table 2. Abstract production rules for the XML-λ Query Language.

of their simpler parts. As the notation we employ the *simply typed lambda calculus*. It is a well-known and formally verified tool for such a purpose.

4.1 Prerequisites

The denotational semantics utilizes a set of functions for the definition of the language meaning. For this purpose, we formulate all necessary mathematical definitions. We start with the data types and specification of the evaluation context followed by the outline of bindings to the $\mathcal{T}_E$ type system. Then, all auxiliary and denotation functions are introduced.

Data Types. Each value computed during the process of the query evaluation is of a type from $Type$. Let E be a type from the type system $\mathcal{T}_E$, we define $Type$ as:

$$\begin{aligned}
&Type ::= BaseType \mid Seq(Type)\\
&Seq(Type) ::= \bot \mid BaseType \times Seq(Type)\\
&BaseType ::= E \mid PrimitiveType\\
&PrimitiveType ::= Boolean \mid String \mid Number
\end{aligned}$$

Primitive types, $Boolean$, $String$, and $Number$, are defined with their set of allowed values as usual. The type constructor Seq stands for the the type of ordered sequences of elements of values of the type $Type$[3]. We use it only for base types, so $Seq(Type)$ is the type of all ordered sequences of elements of base types. We do not permit sequences of sequences. The symbol $\bot$ stands for the empty sequence of types – represents an unknown type. More precisely, we interpret types as algebraic structures, where for each type $\tau \in Type$ there is exactly one carrier $\mathcal{V}_\tau$, whose elements are the values of the respective type τ.

[3] We suppose usual functions nil, cons, append, null, head, and tail for sequences.

Variables. An XML-λ query can use an arbitrary (countable) number of variables. We model variables as pairs $name : \tau$, where $name$ refers to a variable name and τ is the data type of the variable – any member of $Type$. Syntactically, variable name is always prepended by the dollar sign. Each expression in XML-λ has a recognizable type, otherwise both the type and the value are undefined.

Query Evaluation Context. During the process of query evaluation we need to store variables inside a working space known as a context. Formally, we denote this context as the $State$. We usually understand a state as the set of all active objects and their values at a given instance. We denote the semantic domain $State$ of all states as a set of all functions from the set of identifiers $Identifier$ into their values of the type $\tau \in Type$. Obviously, one particular state $\sigma : State$ represents an immediate snapshot of the evaluation process; i.e., values of all variables at a given time. We denote this particular value for the variable x as $\sigma[\![x]\!]$. Simply speaking, the state is the particular valuation of variables. We use the functor $f[x \leftarrow v]$ for the definition of a function change in one point x to the value v.

4.2 Signatures of semantic functions

Having defined all necessary prerequisites and auxiliary functions (recalling that the $SeqType$ represents any permitted type of value), we formalize semantic functions over semantic domains as:

Sem_{Query}	$: Query \to Seq(XML_{Doc}) \to Seq(Type)$
$Sem_{Options}$	$: Options \to (State \to State)$
Sem_{Expr}	$: Expression \to State \to Seq(Type)$
Sem_{Term}	$: Term \to (State \to Boolean)$
Sem_{Frag}	$: Fragment \to State \to Seq(Type)$
Sem_{Assign}	$: Fragment \times Identifier \to State \to State$
$Sem_{RelOper}$	$: Fragment \times RelOperator \times Fragment \to \to (State \to Boolean)$
$Sem_{BinOper}$	$: Term \times BinOperator \times Term \to \to (State \to Boolean)$
$Sem_{AttrCons}$	$: AttrConstr \times State \to Seq(Type)$
$Sem_{ElemCons}$	$: ElemConstr \times State \to Seq(Type)$

Table 3. Semantic functions arities.

4.3 Semantic equations

We start with the semantic equations for the expressions, then we will continue with the semantics of queries.

Terms. Terms are logical expressions. It means, that the meaning of any term in a given state is the Boolean value.

$$Sem_{Term} : Term \rightarrow State \rightarrow Boolean$$

Terms are constructed with help of relational operators (we call it *filter*, but we do not distinguise it here), binary operators, negation sign, or primitive Boolean literals (denoted as B in the definition).

$Sem_{Term}[\![B]\!] = \lambda\sigma.bool[\![B]\!]$
if B is a constant of the type *Boolean*

$Sem_{Term}[\![F_1\ RelOp\ F_2]\!] =$
$\lambda\sigma.Sem_{RelOper}[\![F_1\ RelOp\ F_2]\!]\sigma$

$Sem_{Term}[\![\texttt{'not'}\ T]\!] = \lambda\sigma.not(Sem_{Term}[\![T]\!]\sigma)$

$Sem_{Term}[\![T_1\ BinOp\ T_2]\!] =$
$= \lambda\sigma.Sem_{BinOper}[\![T_1\ BinOp\ T_2]\!]\sigma$

Table 4. Semantic equations for terms.

Relational Operators. Relational operators can be applied to any two fragments and the meaning of resulting expression is the mapping from the current state to Boolean values. They serve in filters.

$$Sem_{RelOper} : Fragment \times RelOperator \times Fragment \rightarrow \rightarrow State \rightarrow Boolean$$

$Sem_{RelOper}[\![F_1\ \texttt{'<'}\ F_2]\!] =$
$= \lambda\sigma.(Sem_{Frag}[\![F_1]\!]\sigma < Sem_{Frag}[\![F_2]\!]\sigma)$

$Sem_{RelOper}[\![F_1\ \texttt{'=='}\ F_2]\!] =$
$= \lambda\sigma.(Sem_{Frag}[\![F_1]\!]\sigma == Sem_{Frag}[\![F_2]\!]\sigma)$

$Sem_{RelOper}[\![F_1\ \texttt{'>'}\ F_2]\!] =$
$= \lambda\sigma.(Sem_{Frag}[\![F_1]\!]\sigma > Sem_{Frag}[\![F_2]\!]\sigma)$

$Sem_{RelOper}[\![F_1\ \texttt{'<='}\ F_2]\!] =$
$= \lambda\sigma.(Sem_{Frag}[\![F_1]\!]\sigma <= Sem_{Frag}[\![F_2]\!]\sigma)$

$Sem_{RelOper}[\![F_1\ \texttt{'>='}\ F_2]\!] =$
$= \lambda\sigma.(Sem_{Frag}[\![F_1]\!]\sigma >= Sem_{Frag}[\![F_2]\!]\sigma)$

$Sem_{RelOper}[\![F_1\ \texttt{'!='}\ F_2]\!] =$
$= \lambda\sigma.(not(Sem_{Frag}[\![F_1]\!]\sigma = Sem_{Frag}[\![F_2]\!]\sigma))$

Table 5. Semantic equations for relational operators.

Binary Operators. Binary operators can be applied to any two terms and the meaning of resulting expression is the mapping from the current state to Boolean values. XML-λ uses clasical logical conectives – logical '`or`' and logical '`and`'

$$Sem_{BinOper} : Term \times BinOperator \times Term \rightarrow State \rightarrow Boolean$$

$Sem_{BinOper}[\![T_1\ \texttt{'or'}\ T_2]\!] =$
$= \lambda\sigma.(Sem_{Term}[\![T_1]\!]\sigma\ or\ Sem_{Term}[\![T_2]\!]\sigma)$

$Sem_{BinOper}[\![T_1\ \texttt{'and'}\ T_2]\!] =$
$= \lambda\sigma.(Sem_{Term}[\![T_1]\!]\sigma\ and\ Sem_{Term}[\![T_2]\!]\sigma)$

Table 6. Semantic equations for binary operators.

Fragments. Fragments are logical parts of filters and have the same meaning as expressions.

$$Sem_{Frag} : Fragment \rightarrow State \rightarrow Seq(Type)$$

$Sem_{Frag}[\![Id]\!] = \lambda\sigma.\sigma[\![Id]\!]$

$Sem_{Frag}[\![f(E_1, ..., E_n)]\!] =$
$= \lambda\sigma.f(Sem_{Expr}[\![E_1]\!]\sigma, ..., Sem_{Expr}[\![E_n]\!]\sigma)$

$Sem_{Frag}[\![F\ P]\!] =$
$= \lambda\sigma.(Sem_{Frag}[\![F]\!] \circ Sem_{Frag}[\![P]\!])\sigma$

$Sem_{Frag}[\![(subquery)(arg)]\!] =$
$= \lambda\sigma.(Sem_{Expr}[\![subquery]\!](\sigma)(Sem_{Expr}[\![arg]\!](\sigma)))$

$Sem_{Frag}[\![I_1 I_2 ... I_n E]\!] =$
$= Sem_{Expr}[\![I_2 ... I_n E]\!](\sigma[Sem_{Expr}[\![E]\!]\sigma \leftarrow I_1])$

$Sem_{Frag}[\![N]\!] =$
$= \lambda\sigma.num[\![N]\!]$ if N is a constant of the type *Numeral*

$Sem_{Frag}[\![S]\!] =$
$= \lambda\sigma.str[\![S]\!]$ if S is a constant of the type *String*

$Sem_{Frag}[\![B]\!] =$
$= \lambda\sigma.bool[\![B]\!]$ if B is a constant of the type *Boolean*

Table 7. Semantic equations for fragments.

Assignments. An assignment expression is a mandatory part of a query. It sets the initial context of the evaluation, more precisely, such expression evaluates a nullary function and stores the result into a variable. Then, the evaluation process will filter results and than iterates over all values and computes remaining results.

$$Sem_{Assign} : Fragment \times Identifier \rightarrow State \rightarrow State$$

$Sem_{Assign}[\![Id\ \texttt{'='}\ F]\!] = \lambda\sigma.\sigma[Id \leftarrow Sem_{Frag}[\![F]\!]\sigma]$

Table 8. Semantic equation for assignments.

Expressions. Each expression e has a defined value $Sem_{Expr}[\![e]\!](\sigma)(\xi)$ in a state σ and in an environment ξ. The state represents the values of variables, the environment represents XML document that is elaborated. The result is a state $Sem_{Expr}[\![e]\!](\sigma)(\xi)$, where all interesting values are bound into local variables. We do not need the environment in the next computation, because all information is in the state. We can model input data as a mapping Env from identifiers to XML documents, formally:

$$Env : Identifier \to XML_{doc}$$

$$Sem_{Expr} : Expression \times State \times Env \to Seq(Type)$$

$$\begin{aligned}
&Sem_{Expr}[\![A_1A_2...A_nF_1F_2...F_m]\!]\sigma\xi = \\
&\quad = Sem_{Expr}[\![A_2...A_nF_1F_2...F_m]\!](Sem_{Assign}[\![A_1]\!]\sigma\xi) \\
&Sem_{Expr}[\![F_1F_2...F_m]\!]\sigma\xi = \\
&\quad = append(Sem_{Expr}[\![F_1]\!]\sigma\xi, Sem_{Expr}[\![F_2...F_m]\!]\sigma\xi) \\
&\quad \text{ if } Sem_{RelOper}[\![F_1]\!]\sigma\xi \ = \ true \\
&Sem_{Expr}[\![F_1F_2...F_m]\!]\sigma\xi = \\
&\quad = Sem_{Expr}[\![F_2...F_m]\!]\sigma\xi \\
&\quad \text{ if } Sem_{RelOper}[\![F_1]\!]\sigma\xi \ = \ false \\
&Sem_{Expr}[\![]\!]\sigma\xi = nil
\end{aligned}$$

Table 9. The semantics of general expressions.

Constructors. Resulting values are created by constructors. A constructor is a list of items which can be either a variable identifier or a constructing expression.

$$Sem_{Cons} : Constructor \times State \to Seq(Type)$$

$$\begin{aligned}
&Sem_{Cons}[\![E_1E]\!]\sigma = append(Sem_{ElemCons}[\![E_1]\!]\sigma, \\
&\quad Sem_{Cons}[\![E]\!]\sigma) \\
&Sem_{Cons}[\![I_1E]\!]\sigma = cons(\sigma[\![I_1]\!], Sem_{Cons}[\![E]\!]\sigma) \\
&Sem_{Cons}[\![\,]\!]\sigma = nil
\end{aligned}$$

Table 10. Semantic equations for constructors.

Element Constructors. The most common kind of resulting value is undoubtedly the element constructor; obviously, all its alternatives are supported – either with empty content, textual content or more complex (i.e. "mixed") content. For all cases we can also attach attributes to elements. In the definition we use abstract functions *element* and *attribute*, which serves to construct output XML values from arguments.

$$Sem_{ElemCons} : ElemConstr \times State \to Seq(Type)$$

Attribute Constructors. Elements can have attributes assigned by attribute constructors

$$Sem_{AttrCons} : AttrConstr \times State \to Seq(Type)$$

$$\begin{aligned}
&Sem_{ElemCons}[\![NA_1...A_nI]\!]\sigma = \\
&= element(N, \sigma[\![I]\!], Sem_{AttrCons}[\![A_1]\!]\sigma, ..., \\
&\quad Sem_{AttrCons}[\![A_n]\!]\sigma) \\
&Sem_{ElemCons}[\![NA_1...A_nE]\!]\sigma = \\
&= element(N, Sem_{Expr}[\![E]\!]\sigma, Sem_{AttrCons}[\![A_1]\!]\sigma, ..., \\
&\quad Sem_{AttrCons}[\![A_n]\!]\sigma) \\
&Sem_{ElemCons}[\![N\ I]\!]\sigma = element(N, \sigma[\![I]\!], nil) \\
&Sem_{ElemCons}[\![N\ E]\!]\sigma = \\
&\quad = element(N, Sem_{Expr}[\![E]\!]\sigma, nil)
\end{aligned}$$

Table 11. Semantic equations for element constructors.

$$Sem_{AttrCons}[\![N\ I]\!]\sigma = attribute(N, Sem_{Expr}[\![I]\!]\sigma)$$

Table 12. The semantic equation for attribute constructors.

Example. Let us show an example of resulting sequence for the XML-λ constructor

```
lambda book attlist [ title $b ] $a
```

The result is the function

$$\lambda\sigma.element(\texttt{'book'}, \sigma[\![a]\!], attribute(\texttt{'title'}, \sigma[\![b]\!]))$$

returning in the given state the string

```
element(book,"the value of a",
  attribute(title,"the value of b"))
```

Options. The only allowed option in the language is now the specification of input XML documents.

$$Sem_{Options} : Options \times Env \to Env$$

$$\begin{aligned}
&Sem_{Options}[\![]\!](E) = E \\
&Sem_{Options}[\![\,\texttt{xmldata(}\ X\ \texttt{)}\ Y]\!] = \\
&\quad = \lambda\xi.Sem_{Options}[\![Y]\!](\xi[\mathcal{D}om(X) \leftarrow X\#])
\end{aligned}$$

Table 13. Semantic equations for options.

We explore a function $\mathcal{D}om(X)$ that converts an input XML document X into its internal representation accessible under identification $X\#$.

Query. A query (denoted as Q) consists of query options (denoted as O), where input XML documents are bound to its formal names, the query expression to be evaluated (denoted as E), and the output construction commands (denoted as C). At first, input files are elaborated, than an initial variable assignment takes place, followed by evaluation of expression. Finally, the output is constructed. This idea is inbuilt into the definition of $Sem_{Query}[\![O\ C\ E]\!]$ bellow. The whole meaning

$Sem_{Query}[\![Q]\!]$ of a query Q can be modeled as a mapping from the sequence of input XML documents into a sequence of output values of the type of $Type$. The definition is carried by the structural induction on the possible forms of an input sequence.

$$Sem_{Query}[\![Q]\!] : Seq(XML_{Doc}) \to Seq(Type)$$

$$Sem_{Query}[\![O\ C\ E]\!] : XML_{Doc} \to Seq(Type)$$

$$\boxed{\begin{array}{l} Sem_{Query}[\![Q]\!](nil) = nil \\ Sem_{Query}[\![Q]\!](cons(H,T)) = \\ \quad = append(Sem_{Query}[\![Q]\!](H), SemQuery[\![Q]\!](T)) \\ Sem_{Query}[\![O\ C\ E]\!] = \\ \quad = \lambda\delta.(Sem_{Cons}[\![C]\!](Sem_{Expr}[\![E]\!](\lambda\sigma.\bot) \\ \quad (Sem_{Options}[\![O]\!](\lambda\xi.\bot)(\delta))) \end{array}}$$

Table 14. Semantic equations for queries.

5 The example

The following example illustrates the computations performed in order to evaluate given XML-λ queries inside a virtual machine. It just computes a simple numerical term. More complex examples could be found in [4].

5.1 Simple computation

Let us suppose the following simple query in the XML-λ and its evaluation.

```
lambda $v1 ($v1 = plus(3, 2))
```

We can compute its meaning according to the XML-λ semantics as (the result is independent on the input XML documents, so we can use the empty sequence *nil* as the input):

$$\begin{array}{l} Sem_{Query}[\![\texttt{'lambda \$v1 (\$v1=plus(3, 2)'}]\!](nil) = \\ = \lambda\delta.(Sem_{Cons}[\![\texttt{'lambda \$v1 (\$v1 = plus(3, 2))'}]\!] \\ \quad (Sem_{Expr}[\![\,]\!])(\lambda\sigma.\bot)(\delta))(nil) = \\ = Sem_{Cons}[\![\texttt{'lambda \$v1 (\$v1 = plus(3, 2))'}]\!] \\ \quad (\lambda\sigma.\bot) = \\ = Sem_{Assign}[\![\texttt{'\$v1 = plus(3, 2)'}]\!] \\ \quad (\lambda\sigma.\bot)(\texttt{'\$v1'}) = \\ = \lambda\sigma_1.\sigma_1[\texttt{'\$v1'} \leftarrow Sem_{Frag}[\![\texttt{'plus(3,2)'}]\!] \\ \quad (\sigma_1)](\lambda\sigma.\bot)(\texttt{'\$v1'}) = \\ = (\lambda\sigma.\bot)[\texttt{'\$v1'} \leftarrow Sem_{Frag}[\![\texttt{'plus(3,2)'}]\!] \\ \quad (\lambda\sigma.\bot)](\texttt{'\$v1'}) = \\ = (\lambda\sigma.\bot)[\texttt{'\$v1'} \leftarrow \lambda\sigma_2.plus(Sem_{Expr}[\![\texttt{'3'}]\!] \\ \quad (\lambda\sigma_2.\bot), Sem_{Expr}[\![\texttt{'2'}]\!](\lambda\sigma_2.\bot)) \\ \quad (\lambda\sigma.\bot)](\texttt{'\$v1'}) = \\ = (\lambda\sigma.\bot)[\texttt{'\$v1'} \leftarrow \lambda\sigma_2.plus(num[\![\texttt{'3'}]\!] \\ \quad (\lambda\sigma_2.\bot), num[\![\texttt{'2'}]\!](\lambda\sigma_2.\bot))(\lambda\sigma.\bot)](\texttt{'\$v1'}) = \\ = (\lambda\sigma.\bot)[\texttt{'\$v1'} \leftarrow \lambda\sigma_2.plus(3,2)(\lambda\sigma.\bot)](\texttt{'\$v1'}) = \\ = (\lambda\sigma.\bot)[\texttt{'\$v1'} \leftarrow \lambda\sigma_2.5(\lambda\sigma.\bot)](\texttt{'\$v1'}) = \\ = (\lambda\sigma.\bot)[\texttt{'\$v1'} \leftarrow \lambda\sigma.5](\texttt{'\$v1'}) = \\ = (\lambda\sigma.\bot)[\texttt{'\$v1'} \leftarrow 5](\texttt{'\$v1'}) = \\ = 5 \end{array}$$

6 Conclusion

In this paper, we have presented syntax and denotational semantics of the XML-λ Query Language, a query language for XML based on simply typed lambda calculus. We use this language within the special XML-λ Framework as an intermediate form of XQuery expressions for description of its semantics. Nevertheless the language in its current version does not support all XML features, e.g. comments, processing instructions, and deals only with type information available in DTD, it can be successfully utilized for fundamental scenarios both for standalone query evaluation or as a tool for XQuery semantics description.

References

1. H. Barendregt: *Lambda calculi with types.* In: Handbook of Logic in Computer Science, Volumes 1 (Background: Mathematical Structures) and 2 (Background: Computational Structures), Abramsky & Gabbay & Maibaum (Eds.), Clarendon, volume 2, Oxford University Press, 1992.
2. T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau: *Extensible markup language (XML) 1.0 (fifth edition)*, November 2008. http://www.w3.org/TR/2008/REC-xml-20081126.
3. D. Draper, P. Fankhauser, M. Fernández, A. Malhotra, K. Rose, M. Rys, J. Siméon, and P. Wadler: XQuery 1.0 and XPath 2.0 formal semantics, September 2005. http://www.w3.org/TR/xquery-semantics.
4. P. Loupal. *XML-λ: A functional framework for XML.* Ph.d. Thesis, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague, February 2010, submitted.
5. J. Pokorný: *XML functionally.* In: B. C. Desai, Y. Kioki, and M. Toyama, (Eds), Proceedings of IDEAS2000, IEEE Computer Society, 2000, 266–274.
6. J. Pokorný: *XML-λ: an extendible framework for manipulating XML data.* In: Proceedings of BIS 2002, Poznan, 2002, 160–168.
7. P. Šárek: *Implementation of the XML lambda language.* Master's thesis, Dept. of Software Engineering, Charles University, Prague, 2002.
8. J. Zlatuška: *Lambda-kalkul.* Masarykova univerzita, Brno, Česká republika, 1993.