Denotational Semantics of the XML-λ Query Language*

Pavel Loupal[1] and Karel Richta[2]

[1] Department of Computer Science and Engineering, FEL ČVUT
Karlovo nám. 13, 121 35 Praha 2
loupalp@fel.cvut.cz

[2] Department of Software Engineering MFF UK,
Malostranske nam. 25, 118 00 Praha 1
richta@ksi.mff.cuni.cz

Abstract. In this paper, we define formally the XML-λ Query Language, a query language for XML, that employs the functional data model. We describe its fundamental principles including the abstract syntax and denotational semantics. The paper basically aims for outlining of the language scope and capabilities.

1 Introduction

In this paper, we define formally the XML-λ Query Language, a query language for XML, that employs the functional data model. The first idea for such an attitude was published in [4]. 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 [6].

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). In the second scenario, the language is utilized as an intermediate language for the description of XQuery semantics. In [3] 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.

2 XML-λ Query Language

In this section, we describe the query language XML-λ, that is based on the simply typed lambda calculus. As a formal tool we use the approach published in

* 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 (GAČR) No. GA201/09/0990.

J. Pokorný, V. Snášel, K. Richta (Eds.): Dateso 2010, pp. 139–146, ISBN 978-80-7378-116-3.

Richta's overview of semantics [5]. For listing of language syntax, we use the Extended Backus-Naur Form (EBNF) and for meaning of queries the denotational semantics [5].

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 shown later. Lambda calculus, written also as 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 [7] 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.

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. Then:

1. *variable:* each variable 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 $((T_1, \ldots, T_n) \rightarrow T)$ 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 $((T_1, \ldots, T_n) \rightarrow T)$
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 $(\mathbf{E} \ \rightarrow T)$.

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 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,
$O : Option$	XML-λ options – XML input attachements,
$C : Constructor$	XML-λ constructors of output results,
$E : Expression$	general expressions, yield a $BaseType$ value,
$T : Term$	sort of expression, yield a $BaseType$ value,
$F : Fragment$	sub-parts of a $Term$,
$BinOp : BinOperator$	binary logical operators,
$RelOp : RelOperator$	binary relational operators,
$N : Numeral$	numbers,
$S : String$	character strings,
$Id : Identifier$	strings conforming to the $Name$ syntactic rule in [2],
$NF : Nullary$	identifiers of nullary functions (subset of $Identifier$),
$Proj : Projection$	identifiers for projections (subset of $Identifier$).

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

For description the meaning of each XML-λ query, we use *denotational semantics*. 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

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

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

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 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:

$$\begin{aligned}
&Type ::= \ BaseType \mid SeqType\\
&SeqType ::= \ \perp \mid BaseType \times SeqType\\
&BaseType ::= \ E \mid PrimitiveType\\
&PrimitiveType ::= \ Boolean \mid String \mid Number
\end{aligned}$$

Primitive types, $Boolean$, $String$, and $Number$, are defined with their set of allowed values as usual. The type $SeqType$ is the type of all ordered sequences of elements of base types[3]. We do not permit sequences of sequences. The symbol $\perp$ stands for the empty sequence of types – represents an unknown type. More

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

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.

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 Auxiliary Functions

For the sake of readability improvement, we propose few semantic functions, denoted as *auxiliary*, that should make the denotations more legible. We introduce functions: $isIdent$ — returns *true* iff its argument denotes an variable identifier in a given $State$, $typeOf$ — returns a type of given argument (one type from the $Type$ set), $valueOf$ — returns a typed value of an expression, *bool* — evaluates its argument as a Boolean value, *num* — converts its argument into a numeric value, and *str* — converts its argument into a string value.

Each expression e has a distinguished type in a state σ. The type can depend on the state because an expression can contain variables. This type is available by calling the $typeOf$ semantic function defined in Table 3.

$$typeOf\ :\ Expression\ \times\ State\ \rightarrow\ Type$$

4.3 XML Schema-Specific Functions

For utilization of the features offered by the XML-λ Framework we propose a number of functions working with information available in the type system. These functions help us to access an arbitrary data model instance. An *application* is informally used for accessing child elements of a given one. More formally, it is an evaluation of a T-object specified by its name. A *projection* is generally used for selecting certain items from a sequence. A *nullary function.* A T-nullary function returns all abstract elements from $\mathbf{E}_T$. *Root Element Access* is a shortcut for a common activity in the XML world — accessing the root element of

$$typeOf[\![e]\!](\sigma) = \begin{cases} \perp & \text{if } e \text{ is a nullary fragment} \\ Boolean & \text{if } e \in \nu_{Boolean} \\ & (e \text{ is a constant of the type } Boolean) \\ Numeral & \text{if } e \in \nu_{Numeral} \\ & (e \text{ is a constant of the type } Numeral) \\ String & \text{if } e \in \nu_{String} \\ & (e \text{ is a constant of the type } String) \\ \tau & \text{if } isIdent[\![e]\!](\sigma) \text{ and } \sigma[\![e]\!] : \tau \\ & (e \text{ is a variable of the type } \tau) \\ Boolean & \text{if } e \text{ is a relational fragment (filter)} \\ & \ e_1 \ RelOper \ e_2 \\ Boolean & \text{if } e \text{ is a logical expression} \\ & \ e_1 \ BinOper \ e_2, \text{ or } not \ e_1 \end{cases}$$

Table 3. Types of general expressions

$$\begin{aligned} &app_{XMLDoc} : E \rightarrow SeqType \\ &proj_{XMLDoc} : SeqType \times \tau \rightarrow SeqType \\ &null_{XMLDoc} : E \times T \rightarrow SeqType \\ &root_{XMLDoc} : E \end{aligned}$$

4.4 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

$$\begin{aligned} Sem_{Query} &: Query \rightarrow (XMLDoc \rightarrow SeqType) \\ Sem_{Options} &: Options \rightarrow (State \rightarrow State) \\ Sem_{Expr} &: Expression \rightarrow (State \rightarrow SeqType) \\ Sem_{Term} &: Term \rightarrow (State \rightarrow Boolean) \\ Sem_{Frag} &: Fragment \rightarrow (State \rightarrow SeqType) \\ Sem_{RelOper} &: Fragment \times RelOper \times Fragment \rightarrow (State \rightarrow Boolean) \\ Sem_{BinOper} &: Term \times BinOper \times Term \rightarrow (State \rightarrow Boolean) \end{aligned}$$

4.5 Semantic Equations

We start with the semantic equations for the expressions. Each expression e has a value $Sem_{Expr}[\![e]\!](\sigma)$ in a state σ. The state represents values of variables. The result is a state, where all interesting values are bound into local variables.

Resulting values are created by constructors. A constructor is a list of items which can be variable identifier or constructing expression. Resulting values can be created by element constructors. Elements can have attributes assigned by attribute constructors.

Options and Queries. The only allowed option in the language is now the specification of input XML documents. We explore a function $\mathcal{D}om(X)$ that converts input XML document X into its internal representation accessible under identification $X\#$. A query consists of query options, where input XML documents

$$
\begin{array}{ll}
Sem_{Term}[\![B]\!] & = \lambda\sigma : State.bool[\![B]\!] \text{ if } B \text{ is a constant of the type } Boolean \\
Sem_{Term}[\![F_1\ RelOp\ F_2]\!] & = \lambda\sigma : State.Sem_{RelOper}[\![F_1\ RelOp\ F_2]\!]\sigma \\
Sem_{Term}[\![\texttt{'not'}\ T]\!] & = \lambda\sigma : State.not(Sem_{Term}[\![T]\!]\sigma) \\
Sem_{BinOper}[\![T_1\ \texttt{'or'}\ T_2]\!] & = \lambda\sigma : State.(Sem_{Term}[\![T_1]\!]\sigma\ or\ Sem_{Term}[\![T_2]\!]\sigma) \\
Sem_{BinOper}[\![T_1\ \texttt{'and'}\ T_2]\!] & = \lambda\sigma : State.(Sem_{Term}[\![T_1]\!]\sigma\ and\ Sem_{Term}[\![T_2]\!]\sigma) \\
Sem_{Term}[\![T_1\ BinOper\ T_2]\!] & = \lambda\sigma : State.Sem_{BinOper}[\![T_1\ BinOper\ T_2]\!]\sigma
\end{array}
$$

Table 4. Semantic equations for terms, relational and binary operators

$$
\begin{array}{l}
Sem_{AttrConstr}[\![N\ I]\!]\sigma = attribute(N, Sem_{Expr}[\![I]\!]\sigma) \\
Sem_{ElemConstr}[\![NA_1...A_nI]\!]\sigma = \\
= element(N, \sigma[\![I]\!], Sem_{AttrCons}[\![A_1]\!]\sigma, ..., Sem_{AttrCons}[\![A_n]\!]\sigma) \\
Sem_{ElemConstr}[\![NA_1...A_nE]\!]\sigma = \\
= element(N, Sem_{Expr}[\![E]\!]\sigma, Sem_{AttrCons}[\![A_1]\!]\sigma, ..., Sem_{AttrCons}[\![A_n]\!]\sigma) \\
Sem_{ElemConstr}[\![N\ I]\!]\sigma = element(N, \sigma[\![I]\!], nil) \\
Sem_{ElemConstr}[\![N\ E]\!]\sigma = element(N, Sem_{Expr}[\![E]\!]\sigma, nil) \\
Sem_{Cons}[\![E_1E]\!]\sigma = append(Sem_{ElemCons}[\![E_1]\!]\sigma, Sem_{Cons}[\![E]\!]\sigma) \\
Sem_{Cons}[\![I_1E]\!]\sigma = cons(\sigma[\![I_1]\!], Sem_{Cons}[\![E]\!]\sigma) \\
Sem_{Cons}[\![\ ]\!]\sigma = nil
\end{array}
$$

Table 5. The semantic equation for constructors

$$
\begin{array}{ll}
Sem_{Frag}[\![Null]\!] & = \lambda\sigma : State.null_{XMLDoc}[\![Null]\!] \\
Sem_{Frag}[\![Id]\!] & = \lambda\sigma : State.\sigma[\![Id]\!] \\
Sem_{Frag}[\![f(E_1, ..., E_n)]\!] & = \lambda\sigma : State.f(Sem_{Expr}[\![E_1]\!]\sigma, ..., Sem_{Expr}[\![E_n]\!]\sigma) \\
Sem_{Frag}[\![F\ P]\!] & = \lambda\sigma : State.(Sem_{Frag}[\![F]\!] \circ Sem_{Frag}[\![P]\!])\sigma \\
Sem_{Frag}[\![(subquery)(arg)]\!] & = \lambda\sigma : State.(Sem_{Expr}[\![subquery]\!](\sigma)(Sem_{Expr}[\![arg]\!](\sigma))) \\
Sem_{Frag}[\![I_1I_2...I_nE]\!] & = Sem_{Expr}[\![I_2...I_nE]\!](\sigma[Sem_{Expr}[\![E]\!]\sigma \leftarrow I_1]) \\
Sem_{Frag}[\![N]\!] & = \lambda\sigma : State.num[\![N]\!] \text{ if } N \text{ is a constant of the type } Numeral \\
Sem_{Frag}[\![S]\!] & = \lambda\sigma : State.str[\![S]\!] \text{ if } S \text{ is a constant of the type } String \\
Sem_{Frag}[\![B]\!] & = \lambda\sigma : State.bool[\![B]\!] \text{ if } B \text{ is a constant of the type } Boolean \\
Sem_{Expr}[\![F]\!]\sigma & = Sem_{Frag}[\![F]\!]\sigma
\end{array}
$$

Table 6. Semantic equations for fragments and expressions

$Sem_{Query}[\![O\ C\ E]\!] =$
$= \lambda\delta : XMLDoc.(Sem_{Cons}[\![C]\!](Sem_{Expr}[\![E]\!](Sem_{Options}[\![O]\!](\lambda\sigma.\bot)(\delta)))$
$Sem_{Query}[\![Q]\!](nil) = nil$
$Sem_{Query}[\![Q]\!](cons(H,T)) = append(Sem_{Query}[\![Q]\!](H), SemQuery[\![Q]\!](T))$
$Sem_{Options}[\![\ ]\!] = \lambda\sigma : State.\bot$
$Sem_{Options}[\![\texttt{xmldata(}\ X\ \texttt{)}\ Y]\!] = \lambda\sigma : State.Sem_{Options}[\![Y]\!](\sigma[\mathcal{D}om(X) \leftarrow X\#])$

Table 7. Semantic equations for options and queries

are bound to its formal names, the query expression to be evaluated, and the output construction commands. First, input files are elaborated, than an initial variable assignment takes place, followed by evaluation of expression. Finally, the output is constructed. The whole meaning of a query can be modeled as a mapping from the sequence of input XML documents into a sequence of output values of the type of $Type$.

5 Conclusions

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 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, or 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 (fourth edition), August 2006. `http://www.w3.org/TR/2006/REC-xml-20060816`.
3. 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.
4. J. Pokorný. XML functionally. In B. C. Desai, Y. Kioki, and M. Toyama, editors, *Proceedings of IDEAS2000*, pages 266–274. IEEE Computer Society, 2000.
5. K. Richta and J. Velebil. *Sémantika programovacích jazyku.* Univerzita Karlova, 1997.
6. P. Šárek. Implementation of the XML lambda language. Master's thesis, Dept. of Software Engineering, Charles University, Prague, 2002.
7. J. Zlatuška. *Lambda-kalkul.* Masarykova univerzita, Brno, Česká republika, 1993.