X-Fun: A universal programming language for processing data trees and XML *

Pavel Labath
Commenius University, Bratislava, Slovakia
labath@dcs.fmph.uniba.sk

Joachim Niehren
INRIA Lille

Abstract

We propose X-Fun, a universal functional programming language for transforming data trees, that can express all of the standardized languages for XML processing XPROC, XQUERY, and XSLT in a uniform manner. While being independent of any particular model, X-Fun can be instantiated with XPATH as the query language, and can be implemented on top of any evaluator for the chosen query language. We present a lean formal model for the operational semantics of X-Fun based on the lambda calculus, and provide an in-memory implementation based on top of SAXON's XPATH evaluator. We also discuss compilers from XSLT, XQUERY, and XPROC to X-Fun, which leads us to in-memory implementations of these XML standards with large coverage and competitive efficiency. Last but not least, it should be pointed out that X-Fun is well-suited as a pipeline language for XML processing, while providing a uniform support for use cases from XSLT and XQUERY at the same time.

Keywords: Data trees, queries, logic, databases, programming languages.

1. Introduction

Functional programming is the major paradigm of programming languages proposed for processing data trees, in spite of many competing approaches based on object-oriented or logic programming (see [10, 15] for instance). The idea of processing data trees in functional languages goes back to Standard ML [18] from the eighties, or even to its predecessors. It was pushed forwards in the nineties by languages such as Mozart-Oz [26], and then become mainstream at the beginning of the millennium with the XML programming languages XQUERY [24] and XSLT [13]. More recently, data trees have found a new place in JSON [6], as used in JavaScript and NOSQL programming languages such as JAQL [4].

Two different approaches can be distinguished. The traditional approach is to represent data trees as terms containing strings, and to define functional programs that perform term rewriting based on pattern matching [3, 21]. Database oriented approaches, in contrast, consider data trees as unranked trees with nodes containing string data values. Pattern matching is replaced by path based node selection queries, such as in XPATH [23] or JSONPATH [9], that can navigate through the nodes of data trees, up, down, left and right, while testing logical properties of nodes, such as equality of data values of nodes (data joins), combined by logical connectives such as conjunction, disjunction and negation. This approach has the merit of importing the declarative power of logical database approaches into functional programming languages.

A first and major drawback of query-based functional languages with data trees so far, is that they either have low coverage in theory and practice or else no lean formal model. Theory driven languages are often based on some kind of macro tree transducers [7, 11, 19, 20], which have low coverage, in that they are not closed under function composition [8] and thus not Turing complete (for instance type checking is decidable [17]). The W3C standardized languages XQUERY 3.0 and XSLT 3.0, in contrast, have large coverage in practice (strings operations, data joins, arithmetics, aggregation, etc.) and in theory, since they are closed by function composition and indeed Turing complete [14]. The definitions of these standards, however, are not formalized and cover hundreds of pages, so that they cannot serve as a basis for a formal analysis.

A second drawback is the tower of language approach adopted for standardized XML processing languages. This is the dark side of standardization in language development with many industrial and academic partners, which is otherwise very beneficial. What happened in the case of XML was the development of a separate language for each class of use cases, which all host the XPATH language for querying data trees based on node navigation. XSLT serves for use cases with recursive document transformations such as HTML publishing, while XQUERY was developed for use cases in which XML databases are queried. In terms of the usual select/extract/load paradigm, XQUERY serves for selection and XSLT for extraction. Since the combination of both is needed in most larger application, the XML pipeline language XPROC [12, 27, 28] was developed and standardized again by the W3C. This is yet another functional programming language for processing data trees based on XPATH.

For resolving the above two drawbacks, the question is whether there exists exists a universal functional programming language for processing data trees, that can cover the various different XML standards in a principled manner. It should satisfy the following four properties.

Lean formal model. It should be based on a formal model as simple and general as the lambda calculus.

Support for logical path query. It should support logical path queries for node selection that can navigate over nodes of data trees up, down, left, and right.

Uniform core language. It should be sufficiently general and expressive, in order to serve as a core language for implementing XQUERY, XSLT, and XPROC in a uniform and highly efficient

* This research was supported in part by the grant VEGA 1/0979/12

manner, with large coverage, and on top of any existing XPATH implementation. And finally:

Universal pipeline language. It should be easy to use as an XML pipeline language, in which all steps of can be defined by functions in a flexible manner, including those of XPROC but not limited to these.

An indicator for the existence of a uniform core language for XML processing is that omnipresent Saxon system [25] implements XSLT and XQUERY on a common platform. However, neither is there any formal description of this platform as a programming language, nor does it support the XML pipeline language XPROC so far. Instead, the existing implementations of XPROC, CALABASH [27] and QUIXPROC [12], are based on Saxon's XPATH engine directly.

The recent work from Castagna et. al. [5] gives further hope that our question will find a positive answer. They present an XPATH-based functional programming language with a lean formal model based on the lambda calculus, thus satisfies our first two conditions above, and can serve as a core language for implementing a subset of XQUERY 3.0. We believe, that relevant parts of XSLT and XPROC can also be compiled into this language, even though this is not shown there. The coverage, however, will remain limited, in particular on the XPATH core (priority is given to strengthening type systems). The concrete problem with their approach is that the integration of XPATH is not fully modular, so that one cannot simply rely on existing XPATH implementations with large coverage. This problem is related to the basic idea of their approach, which is the usage of Huet's zippers for modeling nodes of data trees. Zippers annotate nodes of data trees by their context, so that one can always access their parent. Since zippers are too cumbersome to be programmed manually, they cannot be in any pipeline language. Therefore, our last two requirements are not satisfied.

1.1 Contributions

In this paper, we present the first positive answer to this above question based on X-Fun. This is a new purely functional programming language. X-Fun is simply-typed and a higher-order language and it supports the evaluation of path based queries, that select nodes in data trees. The language of path queries and the model of data trees is left as a parameter. In particular, we can choose to instantiate X-Fun with XPATH and the XML data model.

The formal model of the operational semantics of X-Fun is a lambda calculus with a parallel reduction strategy, which is possible due to the absence of imperative data structures. The main novelties of X-Fun are the admission nodes of data trees as language values of type $\mathbf{node}$. The representation of nodes can be freely chosen by the X-Fun evaluator and is hidden from the programmer. Path queries for node selection can be defined as X-Fun functions of type $\mathbf{node} \to [\mathbf{node}]$. In particular, X-Fun can evaluate XPATH queries with variables bound to values in the X-Fun. Furthermore, one can compute XPATH queries dynamically in X-Fun such as any other value.

We argue that X-Fun fulfills all the requirements of a universal pipeline language. The pipeline's steps can be mapped to functions with multiple inputs and outputs. Higher-order functions make such definitions of steps extremely flexible. The steps of a pipeline can be combined by using function application. In particular, all steps of XPROC can be defined in this manner. Calls to external function can be defined through a foreign function interface, for instance in order to call external evaluators for XQUERY and XSLT.

We show that X-Fun can serve as a uniform core language for implementing XQUERY, XSLT and XPROC. In order to do so, we present compilers of all three languages into X-Fun. We also discuss how to implement X-Fun in an in-memory fashion on top of any in-memory XPATH evaluator. In our current implementation, we use Saxon's in-memory XPATH evaluator. Based on our compilers, we thus obtain new in-memory implementations of XQUERY, XSLT and XPROC with large coverage and competitive efficiency. Indeed, our implementation of XPROC outperforms CALABASH in time by a factor of 2. Our implementations of XQUERY and XSLT are already competitive with those of SAXON even though we did not do any opitimizations: they are only by a factor of 2-12 slower at the time being (a great part of which is due to the omission of tree projection).

Outline. In Section 2 we introduce our general model of data trees, alongside its application to XML documents and XPATH queries. The language itself is introduced in Section 3, where we give the description of it syntax, semantics, and provide the first examples of X-Fun programs. The applications of X-Fun to XML document transformation is studied in Sections 4 and 5. We show how X-Fun can be used as a pipeline and as a core language, and provide compilers from other XML processing languages into X-Fun. These sections also include many additional examples of X-Fun programs. Section 6 contains our notes on the implementation of X-Fun and the results of our experiments.

2. Preliminaries

We introduce a general concept of data trees that can be instantiated to the XML data model and recall some basics on query languages for data trees such as XPATH.

2.1 Data values and data trees

We fix an finite set $Char$ whose elements will be called characters. A data value "$c_1 \cdots c_n$" is a word of characters for $c_1, \ldots, c_n \in Char$. We define $String = Char^*$ to be the set of all data values, and `nil`="" to be the empty data value.

Next, we will fix a natural number $k \geq 1$ and introduce data trees in which each node contains exactly k data values with characters in $Char$.

A *node label* is a k-tuple of data values, i.e., an element of $(String)^k$. The set of data trees $\mathcal{T}$ of label size k over $Char$ is the least set that contains all pairs of node labels and sequences of data trees in $\mathcal{T}$. That is, it contains all unranked trees t with the following abstract syntax:

$$t ::= l(t_1, \ldots, t_n), \qquad \text{where } n \geq 0 \text{ and } l \in String^k.$$

It should be noticed that the set of node labels is infinite, but that each node label can be represented finitely.

Any data tree t can be identified with a unique labeled graph $G^t = (Node, child \uplus next\text{-}sibling, label)$ modulo renaming of nodes, where $Node$ is the set of nodes of t in Dewey notation, $child \subseteq Node \times Node$ the child relation on nodes of t, $next\text{-}sibling \subseteq Node \times Node$ the next sibling relation on nodes of t, and $label : Node \to String^k$ a function that maps a node of t to the tuple of its data values. The *descendant* relation of G^t is $child^+$, is *following-sibling* relation is $next\text{-}sibling^+$. There are also the respective inverse relations *parent*, *ancestor*, *previous-sibling*, and *preceeding-sibling*.

2.2 XML data model

For XML data trees we can fix $k = 4$ and $Char$ the set of Unicode characters, and restrict ourselves to node labels of the following forms, where all v_i are data values:

("element", v_1, v_2, nil)
("attribute", v_1, v_2, v_3)
("comment", v_1, nil, nil)
("processing-instruction", v_1, v_2, nil)
("document", nil, nil, nil)
("text", v_1, nil, nil)

An element ("element", v_1, v_2, nil) has three data values: its type "element", a namespace v_1 and a name v_2. An attribute ("attribute", v_1, v_2, v_3) has four data values: its type "attribute", a namespace v_1, a name v_2, and the attribute value v_3. A text node ("text", v_1, nil, nil) contains its type "text" and a data value v_1. Besides these, there are comments, processing instructions and the rooting document node.

2.3 XPATH queries

XPATH is a navigational language for defining queries on XML data trees. For instance, one can use the XPATH expression

```
child::a/descendant::b[ancestor::c and
                       contains(text(),"Jemal")]
```

for navigating from some start node in an XML data tree to child nodes labeled by a, and then to all descendants labeled by b, which are the target nodes, under the condition, that each target node has an ancestor labeled by c and that its text contains "Jemal". The basic relations of XPATH expressions are the axes *child*, *following-sibling*, *descendant*, *ancestor*, etc. The axes of XPATH are basically the relations of data trees except that XPATH axes are restricted by types. Attributes, for instance, cannot be accessed by the child axis, but there is a special axis for accessing them, the *attribute* axis. XPATH expressions can access nodes and documents provided by the environment using the `$` syntax, for instance `$a/child::b` accesses the b children of the node referred to by the variable a.

The XPATH language also defines a suite of functions for manipulation of strings and numbers and can access the node label data values using builtin functions. Although the primary purpose of XPATH is node selection, XPATH also supports expressions which return values of other types instead of a list of nodes. For example, the expression

```
concat(string(child::a), 'ab')
```

takes the string value of the child labeled by a and concatenates it with the string "ab".

3. Language X-Fun

In this section we introduce X-Fun, a new functional programming language for transforming data trees. X-Fun can be applied to all kinds of data trees with a suitable choice of its parameters. We will instantiate the case of data trees satisfying the XML data model concomitant with XPATH as a query language in particular.

We start with introducing the types and values of X-Fun (Section 3.1). Then we discuss how X-Fun variables are mapped to values of the right type by a program environment (Section 3.2). The novel part here is how to specify the type $\mathbf{node}$, i.e., how to model nodes of trees. Next, we explain how to map path queries to X-Fun values, by using particular X-Fun expressions with variables (Section (3.3). The general syntax of X-Fun expressions is given in Section 3.4. It relies on standard concepts of functional programming languages exclusively, but is equipped with special builtin functions, which enable the evaluation of node selection queries on trees. The precise choice of the builtin functions depends on the chosen kind of data trees and path queries, and is thus left as a parameter of X-Fun. The typing rules for X-Fun's simple type system are presented in Section 3.6. The semantics of X-Fun is given by a big step evaluator in Section 3.7. Some syntactic sugar and a first example of an X-Fun program are given in Sections 3.8 and 3.9.

3.1 Types and Values

The X-Fun language supports higher-order values and expressions with the following types:

$$T \;::=\; \mathbf{node} \;\mid\; \mathbf{tree} \;\mid\; \mathbf{char} \;\mid\; \mathbf{pathletter} \;\mid\; T_1 \times \ldots \times T_n \;\mid\; [T] \;\mid\; T_1 \to T_2$$

A value of type $\mathbf{char}$ is an element of $Char$, a value of type $\mathbf{tree}$ is an element of $\mathcal{T}$. A value of type $\mathbf{node}$ will be a node of the graph of one of the trees stored by the environment of the X-Fun evaluator. This collection will contain graphs of all input trees and intermediate trees computed by the evaluator. The precise node identifiers chosen by the evaluator are left internal (to the mapping from trees to graphs).

The special type $\mathbf{pathletter}$ denotes the union of the types $\mathbf{char}$, $\mathbf{node}$, $[\mathbf{node}]$ and $[\mathbf{char}]$, as needed to represent XPATH queries by lists of query letters, that is by values of type $\mathbf{path} = [\mathbf{pathletter}]$.

As usual, we support list types $[T]$ which denotes all lists of values of type T, product types $T_1 \times \ldots \times T_n$, whose values are all tuples of the values of types T_i, and function types $T_1 \to T_2$, whose values are all partial functions of values of type T_1 to values of type T_2. A data value "$c_1 \cdots c_n$" $\in String$ is considered as a list of characters of type $\mathbf{string} = [\mathbf{char}]$. A node label is considered a k-tuple of strings, i.e., as a value of type $\mathbf{label} = \mathbf{string}^k$. Hedges are considered as lists of trees of type $\mathbf{hedge} = [\mathbf{tree}]$.

3.2 Variables and Environments

Expressions of the X-Fun language are constructed from an infinite set of typed variables. A variable is ranged over by x and its type by $\textsc{Type}(x)$. The evaluation of an expression is done in an environment, that is a partial function binding variable x to values of $\textsc{Type}(x)$.

The set of values of type $\mathbf{node}$ can increase during the computation. At any time point, it is defined by the environment, in which the evaluation happens. An environment contains a graph of a bag of trees at any time point, so that the set of values of type $\mathbf{node}$ is the set of nodes of this graph. New trees can be added to the environment at any time point during the evaluation. The order in which new trees are added does not matter. Whenever a tree is added to the environment, a unique identifier for all of its nodes is chosen nondeterministically, while the identifier of the root is returned. The precise choice of this identifier does not matter, since the identifiers are always left internal to the environment. Whenever a node needs to be output to the user, a variable is returned that gives access to the node via the environment.

3.3 XPath queries as X-Fun expressions

We will consider XPath expressions as values of our programming language. This is done in such a manner that the variables in XPATH expressions can be bound to values of the programming language. For instance, if we have an XPATH expression

```
$x//book[auth=$y]
```

The one might want want to evaluate this expression while variable x is bound to a node of some tree and variable y to some data value. In order to make this possible, we consider an XPATH expression as a value of type $[\mathbf{pathletter}]$, i.e., as lists of nodes, characters and strings. The above query will be represented by the following expression of type $[\mathbf{pathletter}]$, where x is a variable of type $\mathbf{node}$ and y a variable of type $\mathbf{string}$:

$$x|\text{'/'}|\text{'/'}|\text{'b'}|\text{'o'}|\text{'o'}|\text{'k'}|\text{'['}|\text{'a'}|\text{'u'}|\text{'t'}|\text{'h'}|\text{'='}|y|\text{']'}|nil$$

The concrete syntax of X-Fun supports syntactic sugar for values of type $\mathbf{path}$, so that the above expression can be defined as:

Expressions

E	::=	
variables	|	x
strings	|	"w", for $w \in \Delta^*$
equality	|	E_1 = E_2
Boolean conditional	|	**if** E **then** E_1 **else** E_2
list construction	|	$E_1 \vert E_2$
list decomposition	|	**match** E { $x_1 \vert x_2$ -> E_1 **nil** -> E_2 }
tuple construction	|	$(E_1, \ldots, E_n)$, $n \geq 2$
tuple decomposition	|	**let** $(x_1, \ldots, x_n)$ = E_1 **in** E_2
function definition	|	**fun** x:T_1->T_2 { E }
function application	|	$E_1(E_2)$
exception handler	|	**try** E_1 **catch**(x) E_2
raise exception	|	**raise**$_T(E)$

Figure 1. Syntax of X-Fun's expressions

```
"$x//book[auth=$y]"
```

In order to support XPATH evaluation by X-Fun, three builtin functions are supported: An evaluator $\text{evalPath}_{[\mathbf{node}]}$ of XPATH expressions returning lists of nodes, of type $[\mathbf{pathletter}] \to [\mathbf{node}]$, an evaluator $\text{evalPath}_{\mathbf{string}}$ of XPATH expressions returning data values, so of type $[\mathbf{pathletter}] \to \mathbf{string}$, and an evaluator $\text{evalPath}_{\mathbf{bool}} : [\mathbf{pathletter}] \to \mathbf{bool}$. In an implementation of X-Fun, these functions can be mapped straightforwardly to existing XPATH evaluators.

It should be noticed that XPATH-like expressions can also be defined for other queries languages and other kinds of data trees, which do not satisfy the XML data model. In this case, the only changes required are the concrete syntax of the queries, and the semantics of the three builtin functions above.

3.4 Syntax of X-Fun expressions

X-Fun is a simply typed purely functional programming language, whose values subsume higher-order function, trees, and strings. The evaluation strategy of X-Fun is fully parallel, which is possible since no imperative constructs are permitted.

The syntax of X-Fun programs E is given in Figure 1. All expressions of X-Fun are standard in functional programming languages. The novel part is limited to operations on nodes of data trees, which are bound to global variables, and their semantics.

We now consider different kinds of subexpressions E of X-Fun programs. A variable x is evaluated to the value of type $\text{TYPE}(x)$, once such a binding is produced. Otherwise, the evaluation of x fails. The expression "w" returns the string w. The equality test E_1 = E_2 enables us to test the equality of two values. It is only defined the types **char**, **node**, and **tree**.

The list constructor $E_1|E_2$ prepends an element to a list. The match expression **match** E { $x_1|x_2$ -> E_1 **nil** -> E_2 } decomposes lists. If the list is non-empty then the first branch is evaluated (with x_1 and x_2 bound to the head and tail of the list, respectively). In the case of an empty list, the second branch is evaluated.

The tuple constructor $(E_1, \ldots, E_n)$ constructs tuples of values, which can then be decomposed by pattern matching within let expressions. We assume that a tuple always contain at least two members, i.e,, there is no tuple without a comma. Parentheses as in the expression (E) are used for determining expression precedence, so that $((E))$ = (E). The expression **let** $(x_1, \ldots, x_n)$ = E_1 **in** E_2 extends the environment with the definitions for $x_1, \ldots, x_n$, and evaluates E_2 in this environment.

A function definition **fun** x:T_1->T_2 { E } returns a new function, with argument $x : T_1$ and return value of type T_2 obtained by the evaluation of the function body E. The expression $E_1(E_2)$ applies a function to a value.

The full language also supports exception handling, where exceptions are values of type **string**. This mechanism is rather standard, so we sketch it only briefly. An exception handler is an expression of the form **try** E_1 **catch**(x) E_2. It attempts to evaluate subexpression E_1 and if this succeeds it returns its value. Otherwise, if an exception string s is raised, then the evaluator continues with the evaluation of E_2, with variable x bound to the exception string. An exception can be raised either if the evaluator fails, for instance when calling a partial function with an invalid argument), or if a raise exception of the form **raise**$_T(E)$ is evaluated. A raise expression annotated by type T can be use in any place, where an expression of type T is expected.

3.5 Builtin operators and library

At the beginning of the evaluation, the environment contains bindings of all global variables given in Figures 2 and 3. The first figure contains builtin functions and values, while the second contains a library of functions defined on top of these in X-Fun.

We start with the builtin operators in Figures 2. The first block contains five, whose semantics are parameters, depending on the query language and data model. For a label l and a sequence of trees h, the function application $\text{makeTree}(l, h)$ returns the data tree $l(h)$, if $l(h)$ is a well-formed data tree (for instance with respect to the XML data model) and raises an exception otherwise. The function $\text{evalPath}_{[\mathbf{node}]}(p)$ evaluates a node selecting path p, while assuming that the start node of p is given by a variable at the beginning of path p. The same holds for function application $\text{evalPath}_{\mathbf{string}}(p)$ and $\text{evalPath}_{\mathbf{bool}}(p)$, except that the path p must return a string or a Boolean, respectively. Whenever p is not well-formed (for instance with respect to the XPATH 3.0 specification) or does not have the correct output type, an error is raised. Note that path expressions are X-Fun values, which means they can be computed dynamically by the X-Fun program using information from the input data tree.

The next six operators are generic and do not depend on the specific kind of data trees. The variable nil_T refers to the empty list of type T and the variables true and false refer to the Booleans. A function application $\text{subtree}(v)$ returns the subtree rooted at node v. A function application $\text{label}(v)$ returns the label of node v (which we can assume to be well-formed, since we will add only well-formed trees to the environment). The function $\text{root}_{\mathbf{tree}}$ returns the identifier of the root node of the tree, and is used for storing the graph of the tree in the environment. This function can be used to access nodes of newly generated trees, by starting path navigation from their root.

We next discuss functions of the X-Fun library in Figure 3 and their implementation in X-Fun. The first block contains utility functions operating on data trees. These functions are specific to the XML data model, where **label** = **string**[4]. For other kinds of data trees a different set of utility functions may be defined. The function text computes the text value of the subtree rooted at the input node. It is defined as follows:

```
text = fun x:node ->string
          {evalPath_string("string($x)")}
```

Alternatively, it can also be defined without relying on the XPATH `string` function, by a recursive function that concatenates the texts of all descendants of the input node. We provide the following three

Global variable	TYPE	Description
parameters		
makeTree	$\mathbf{label} \times [\mathbf{tree}] \to \mathbf{tree}$	tree constructor
$\text{evalPath}_{\mathbf{string}}$	$\mathbf{path} \to \mathbf{string}$	query evaluator
$\text{evalPath}_{[\mathbf{node}]}$	$\mathbf{path} \to [\mathbf{node}]$	query evaluator
$\text{evalPath}_{\mathbf{bool}}$	$\mathbf{path} \to \mathbf{bool}$	query evaluator
less	$\mathbf{char} \times \mathbf{char} \to \mathbf{bool}$	ordering
fixed		
nil_T	$[T]$	empty lists
true	$\mathbf{bool}$	true
false	$\mathbf{bool}$	false
subtree	$\mathbf{node} \to \mathbf{tree}$	subtree selection
label	$\mathbf{node} \to \mathbf{label}$	label selection
$\text{root}_{\mathbf{tree}}$	$\mathbf{tree} \to \mathbf{node}$	root selection

Figure 2. Builtin operators of X-Fun

Global var.	TYPE	Description
data trees		
text	$\mathbf{node} \to \mathbf{tree}$	text of subtree
element	$\mathbf{string} \times [\mathbf{tree}] \to \mathbf{tree}$	element tree
attribute	$\mathbf{string} \times \mathbf{string} \to \mathbf{tree}$	attribute leaf
textLeaf	$\mathbf{string} \times \mathbf{string} \to \mathbf{tree}$	text leaf
lists		
$\text{map}_{T \to T'}$	$(T \to T') \times [T] \to [T']$	map to all
$\text{mapc}_{T \to [T']}$	$(T \to [T']) \times [T] \to [T']$	map and concat
member_T	$(T \times [T]) \to \mathbf{bool}$	list membership
filter_T	$((T \to \mathbf{bool}) \times [T]) \to [T]$	filter sublist
sort_T	$[T] \times (T \times T \to \mathbf{bool}) \to [T]$	sort lists
group	$[\mathbf{node}] \times (\mathbf{node} \to \mathbf{string}) \to [[\mathbf{node}]]$	group nodes
external		
foreign_T	$\mathbf{string} \to T$	foreign functions

Figure 3. Library

convenience functions to simplify creation of data trees in the most common cases.

```
element = fun (n, h): string×[tree] -> tree
  {makeTree(("element", n, nil_char, nil_char), h)}
attribute = fun (n, d): string×string -> tree
  {makeTree(("attribute", n, nil_char, d), nil_tree)}
textLeaf = fun t: string ->tree
  {makeTree(("text", t, nil_char, nil_char), nil_tree)}
```

The second block in Figure 3 contains various list functions. The usual recursive function $\text{map}_{T \to T'}$ is defined for all types T and T' as follows:

```
map_{T->T'} = fun x: (T->T')×[T] -> [T'] { let
        (f,l) = x
    in match l {
        h|t -> f(h) | map_{T->T'}(f, t)
        nil -> nil_{T'}
        }
    }
```

Here, we assume that $\text{map}_{T \to T'}$ is a distinct variable for any two types T and T', and that $\text{TYPE}(\text{map}_{T \to T'}) = (T \to T') \times [T] \to [T']$. The function $\text{mapc}_{T \to [T']}$ first applies $\text{map}_{T \to [T']}$ and then concatenates the elements of its output (of type $[[T']]$). Function member_T checks the membership of a value of type T in a list. The function sort_T sorts the list with elements of type T according to an ordering of type $(T \times T) \to \mathbf{bool}$. The function group takes a list of nodes and function mapping nodes to strings. It then produces a list of lists of nodes by grouping nodes mapping to the same text together.

The last block provides a foreign function interface through global variables foreign_T of type $\mathbf{string} \to T$. This interface allows to call functions of the implementation's host language, which is Java in our case. This gives us access to the SAXON engine, for instance for calling XSLT or XQUERY evaluators.

3.6 Typing

The X-Fun language is based on simple typing without polymorphism. We do not try to obtain exact typing rules for XPATH expressions (as with semantic subtyping [5]), since this is not supported by most existing XPATH evaluators.

The typing rules of X-Fun are explained in Figure 4. Most of them are standard. However, it should be noticed that we require explicit type annotations on function definitions and when raising exceptions, since this type cannot always be inferred otherwise.

The rules for the union type $\mathbf{pathletter}$ state that the types $\mathbf{char}$, $\mathbf{node}$, $[\mathbf{node}]$ and $\mathbf{string}$ are subtypes of $\mathbf{pathletter}$. Therefore any list of type $[\mathbf{char}]$ is also of type $[\mathbf{pathletter}]$, but not conversely.

A complete program is an expression of function type $T_1 \to T_2$, for which all free variables are bound by the original environment, such as those in Figure 2.

The arguments of the function will be provided when the function is called. This is possible only if T_1 does not require to pass any nodes as input, since node identifiers cannot be known to the user. Similarly, whenever a node is output according to type T_2, only a variable is returned that is bound to the node in the environment. One can then use the environment, in order to access more information about the node via the variable.

3.7 Semantics

We define the semantics of X-Fun by the small-step evaluator in Figure 5, which rewrites expressions to expression or values in a given environment.

What is new compared to standard λ-calculi is the way in which nodes of trees are explored. The intuition is as following. Basically, an X-Fun program is a pipeline of transformation steps composed by function applications. Each step can use the existing trees to compute a new tree that is then add to the environment. Thereby identifiers for the nodes of the tree must be generated, which then become accessible by navigation from the root. Within each step, the evaluator may navigate on the existing trees up, down, left and right, and compute joins between data values of different trees, while constructing the output tree of the step in a top-down manner.

As usual, the evaluator guarantees that lists remain homogeneous (up to the heterogeneity within the union type $\mathbf{pathletter}$) and that expressions will be reduced to a value in a finite number of steps (which is then unique modulo renaming of node identifiers and variables names), run indefinitely, or get stuck with a programming error. For instance, an application $\text{evalPath}_{[\mathbf{node}]}(p)$ cannot proceed if p is a value of type $[\mathbf{pathletter}]$ but does not represent a well-formed path query, or it represents a path query returning a string instead of a list of nodes.

The semantics is defined with respect to an environment D. It defines two kinds of judgments $E \xrightarrow{\eta} E'$ or $E \xrightarrow{\eta} v$ where E and E' are expressions and v is a value with respect to D. The annotation η on the arrow is an action changing the environment D. This action may also be empty, in which case we omit the annotation on the arrow. We consider an environment as an abstract data structure

$$\frac{\textsc{Type}(x) = T}{x : T} \qquad \frac{\mathit{true}}{\text{"}w\text{"} : [\mathbf{char}]} \qquad \frac{E_1, E_2 : T \qquad T \in \{\mathbf{char}, \mathbf{node}, \mathbf{tree}\}}{E_1 = E_2 : \mathbf{bool}} \qquad \frac{E_1 : T \qquad E_2 : [T]}{E_1 | E_2 : [T]}$$

$$\frac{E : [T] \qquad \textsc{Type}(x_1) = T \qquad \textsc{Type}(x_2) = [T] \qquad E_1, E_2 : T'}{\mathbf{match}\ E\ \{\ x_1 | x_2\ \texttt{->}\ E_1\ \mathbf{nil}\ \texttt{->}\ E_2\ \} : T'} \qquad \frac{E_1 : T_1 \times \cdots \times T_n \qquad \textsc{Type}(x_i) = T_i \qquad E_2 : T}{\mathbf{let}\ (x_1, \ldots, x_n) = E_1\ \mathbf{in}\ E_2 : T}$$

$$\frac{E_i : T_i}{(E_1, \ldots, E_n) : T_1 \times \cdots \times T_n} \qquad \frac{E_1 : T \to T' \qquad E_2 : T}{E_1(E_2) : T'} \qquad \frac{\textsc{Type}(x) = T_1 \qquad E : T_2}{\mathbf{fun}\ x : T_1 \texttt{->} T_2\ \{\ E\ \} : T_1 \to T_2}$$

$$\frac{E : \mathbf{bool} \qquad E_1, E_2 : T}{\mathbf{if}\ E\ \mathbf{then}\ E_1\ \mathbf{else}\ E_2 : T} \qquad \frac{E_1 : T \qquad E_2 : \mathbf{string} \to T}{\mathbf{try}\ E_1\ \mathbf{catch}\ E_2 : T} \qquad \frac{E : \mathbf{string}}{\mathbf{raise}_T(E) : T}$$

$$\frac{E : \mathbf{char}}{E : \mathbf{pathletter}} \qquad \frac{E : \mathbf{node}}{E : \mathbf{pathletter}} \qquad \frac{E : [\mathbf{node}]}{E : \mathbf{pathletter}} \qquad \frac{E : \mathbf{string}}{E : \mathbf{pathletter}}$$

Figure 4. Typing rules of X-Fun.

$$\frac{E \xrightarrow{\eta} E' \qquad C[.]\ \text{evaluation context} \qquad \eta\ \text{action on D}}{C[E] \xrightarrow{\eta} C[E']} \qquad \frac{v = \mathit{getVal}^D(x)}{x \to v} \qquad \frac{v \neq v'\ \text{values}}{v = v' \to \mathrm{false}}$$

$$\frac{v\ \text{value}}{v = v \to \mathrm{true}} \qquad \frac{e_i\ \text{is a value} \qquad 1 \le i \le n}{\mathbf{let}\ (x_1, \ldots, x_n) = (e_1, \ldots, e_n)\ \mathbf{in}\ E \xrightarrow{\mathit{bindVar}_D(x, e_i)} \mathbf{let}\ (\ldots, x_{i-1}, x_{i+1}, \ldots) = (\ldots, e_{i-1}, e_{i+1}, \ldots)\ \mathbf{in}\ E}$$

$$\frac{\mathit{true}}{\mathbf{let}\ () = ()\ \mathbf{in}\ E \to E} \qquad \frac{\mathit{true}}{\mathbf{match}\ E | E'\ \{\ x | x'\ \texttt{->}\ E_1\ \mathbf{nil}\ \texttt{->}\ E_2\ \} \to \mathbf{let}\ (x = E, x' = E')\ \mathbf{in}\ E_1}$$

$$\frac{\mathit{true}}{\mathbf{match}\ \mathrm{nil}_T\ \{\ x | x'\ \texttt{->}\ E_1\ \mathbf{nil}\ \texttt{->}\ E_2\ \} \to E_2} \qquad \frac{\mathit{true}}{\mathbf{fun}\ x : T_1 \texttt{->} T_2\ \{\ E\ \}(E') \to \mathbf{let}\ x' = E'\ \mathbf{in}\ E[x/x']}$$

$$\frac{\mathit{true}}{\mathbf{if}\ \mathrm{true}\ \mathbf{then}\ E_1\ \mathbf{else}\ E_2 \to E_1} \qquad \frac{\mathit{false}}{\mathbf{if}\ \mathrm{false}\ \mathbf{then}\ E_1\ \mathbf{else}\ E_2 \to E_2} \qquad \frac{t = \mathit{subtree}^D(v)}{\mathrm{subtree}(v) \to t} \qquad \frac{t\ \text{is tree}}{\mathrm{root}_{\mathbf{tree}}(t) \xrightarrow{v = \mathit{addTree}^D(t)} v}$$

$$\frac{l(h)\ \text{valid data tree of the chosen data model}}{\mathrm{makeTree}((l, h)) \to l(h)} \qquad \frac{v = \mathit{evalPath}_T^D(w)}{\mathrm{evalPath}_T(w) \to v}$$

Figure 5. Small-step semantics of X-Fun expressions in an environment D.

that implements the graph of a bag of data trees and a binding of variables with respect to this graph. Besides the empty action, we support the following kinds of actions η on environments D, where x is a variable, and v are values with respect to D:

$\mathit{bindVar}^D(x, v)$	bind variable x to value v
$v = \mathit{getVal}^D(x)$	get value v bound to x
$v = \mathit{addTree}^D(t)$	add a copy of tree t to the graph and return the identifier v chosen for its root

Furthermore, the environment has the ability to evaluate path queries, defining the following two kinds of judgments, where v and v' are values, t is a data tree, and T is one of the three output types for path evaluators ($[\mathbf{node}]$, $\mathbf{node}$, or $\mathbf{string}$):

$t = \mathit{subtree}^D(v)$	return the subtree t a node v
$v' = \mathit{evalPath}_T^D(v)$	return value v' of evaluating path v

The precise definitions of $\mathit{evalPath}_T^D$ is the first parameter of the X-Fun language. The second is the definition of what is a valid data tree in the data model under consideration.

An evaluation context $C[.]$ is an X-Fun expression E that contains a single occurrence of a fresh variable . that we call the hole marker. As usual for parallel evaluation, the hole marker can be anywhere, but not in the body E of some function definition $\mathbf{fun}\ x : T_1 \texttt{->} T_2\ \{\ E\ \}$, and not in the continuations E_1 or E_2 of a conditional expression $\mathbf{if}\ E\ \mathbf{then}\ E_1\ \mathbf{else}\ E_2$, or a match expression $\mathbf{match}\ E\ \{\ x_1 | x_2\ \texttt{->}\ E_1\ \mathbf{nil}\ \texttt{->}\ E_2\ \}$ We will write $C[E']$ for the expression obtained by substituting the unique occurrence of the hole marker . by E.

We now discuss the rules of the operational semantics in Figure 5. Note that we assume as usual, that all bound variables are always renamed apart before the application of any of these rules. The first rule states that an evaluation step can be applied to any subexpression in an evaluation context. Second, a variable can be replaced by the value to which it is bound in the environment, if there is any. Third and fourth, equality tests, reduce to false on distinct value, and to true on equal values. The rule for **let** expressions states, that a let binding $x_i = v_i$ of a variable to a value can be removed when adding this binding to the environment. When all the bindings of a **let** expression are removed, the whole expression can be replaced by the **let** body. The rules for **match** and **if** expressions decompose lists as usual. The reduction of function applications $\mathbf{fun}\ x : T_1 \texttt{->} T_2\ \{\ E\ \}(E')$ creates a let expression $\mathbf{let}\ x' = E'\ \mathbf{in}\ E[x/x']$ binding a fresh variable x' to the argument E', while starting the evaluation of the body E of the function, but with x replaced by x'.

Applications of builtin functions will be reduced as follows: subtree(v) returns the subtree of node v in the graph of D, while

$\text{root}_{\textbf{tree}}(t)$ adds tree t to environment D and returns the root node that was created thereby.

Only the last 2 reduction rules depend on the parameters of X-Fun. An application $\text{makeTree}((l,h))$ is reduced to the data tree $l(h)$ if the latter is well-formed in the chosen data model. Therefore, we can assume that all data trees in the environment D of the evaluator are always well-formed. An application $\text{evalPath}_T(w)$ is reduced to the result of evaluating query w with respect to D, i.e. $evalPath_T^D(w)$, if it exists.

3.8 Syntactic sugar

In the X-Fun snippets in the rest of the paper we shall employ a number of syntactic shortcuts, which enable us to express more succinctly some X-Fun construct. These are:

list concatenation The binary operator `*` shall concatenate two lists of the same type.

equality of lists/tuples We shall extend operator = to operate on lists and tuples (not containing nodes or functions) in the obvious way.

let with multiple definitions We shall usually write **let** $x_1 = E_1, \ldots, x_n = E_n$ instead of **let** $(x_1, \ldots, x_n) = (E_1, \ldots, E_n)$.

multi-argument function We shall write **fun** $(a,b):T_1 \times T_2\text{->}T$ instead of **fun** $x:T_1 \times T_2\text{->}T$ followed by a **let** $(a,b) = x$

3.9 Examples

In Figure 6 we illustrate a transformation that converts an address book into HTML, while leaving out secret information. The address fields are assumed to be unordered in the input data tree, while the fields of the output HTML addresses should be published in the order `name`, `street`, `city`, `phone` and `email`.

An X-Fun program defining this transformation is given in Figure 7. Starting at the root, if first locates all address records, and applies the function `convert_address` to each of them. For each address record, the program first extracts the values of the fields `name`, `street`, and `city` located at some children of `x`. These values are then bound to string variables named alike. In the scope of these variables, the function `convert_email` is defined, which outputs a hyperlink by which one can send email to this address, while displaying the `name` too.

4. X-Fun as a pipeline language

XML documents are often processed in multiple stages. A schematic representation of an example pipeline is given in Figure 8. The individual steps in such a pipeline can be schema validation, transformation with XSLT or XQUERY, etc. X-Fun can be easily used to express multi-stage processing of documents. Each stage of the processing can be expressed by a function taking one or more data trees as input, and returning one or more data trees as its output. The processing stages are then simply linked using function composition, and variables can be used to store trees flowing through the pipeline and passing them to the correct steps. The example in Figure 8 can be expressed in X-Fun as follows:

```
fun input: tree->tree {
  let doc1 = step1(input) in
  step3(step2a(doc), step2b(doc))
}
```

Steps with multiple inputs and output can dealt with by using functions with input and output tuples.

XPROC is an XML-based language for describing tree transformation as a sequence of steps. Its main feature is the ability to

```
fun book : tree->tree { let
 bookroot = root_tree(book) in let
 convert_address = fun x : node->tree { let
   (name, street, city) =
     (evalPath_string("string($x/child::name)"),
      evalPath_string("string($x/child::street)"),
      evalPath_string("string($x/child::city)")
     ) in let
   convert_email = fun x : node->tree {
     element(
       "a",
       [attribute("href", "mailto:" * name
              * "<" * text(x) * ">"),
              textLeaf(text(x))])
   }
 in
   element("li", [
     element("p", [textLeaf(name)]),
     element("p", [textLeaf(street)]),
     element("p", [textLeaf(city)])]
     * map_node->tree(
         fun x : node->tree {
           element("p", [
             textLeaf("Phone: " * text(x))])
         },
         evalPath_[node]("$x/phone[not(secret)]"))
     * map_node->tree(convert_email,
       evalPath_[node]("$x/child::email")))
 }
in
  element("ol",
    map_node->tree(convert_address,
      evalPath_[node]("$book/descendant::address")))
}
```

Figure 7. X-Fun program converting address books to HTML.

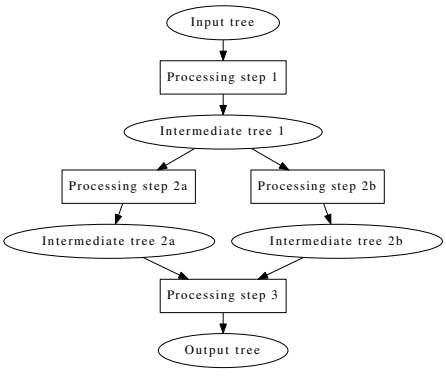

Figure 8. A pipeline example

```
<addresses>
  <address>
    <name>Jemal Antidze</name>
    <phone><secret/>99532 305072</phone>
    <email>jeantidze@yahoo.com</email>
    <phone>99532 231231</phone>
    <email>jeantidze@ip.osgf.ge</email>
    <city>Tblissi</city>
  </address>
  <address>
    <name>Joachim Niehren</name>
    <city>Lille</city>
    <street>Rue Esquermoise</street>
  </address>
</addresses>
```

⇒

```
<ol>
  <li>
    <p> Jemal Antidze </p>
    <p> Tblissi <p/>
    <p> Phone: 99532 231231 <p/>
    <a href="mailto: Jemal Antidze
       &lt; jeantidze@yahoo.com &gt;">
       jeantidze@yahoo.com </a>
    <a href="mailto: Jemal Antidze
       &lt; jeantidze@ip.osgf.ge &gt;">
       jeantidze@ip.osgf.ge </a>
  </li>
  <li>
    <p> Joachim Niehren </p>
    <p> Rue Esquermoise </p>
    <p> Lille </p>
  </li>
</ol>
```

Figure 6. Publication of an address book in HTML except for secret entries.

```
<p:split-sequence name="split" test="//b">
  <p:input port="source" select="//a"/>
</p:split-sequence>

<p:pack wrapper="pair">
  <p:input port="alternate">
    <p:pipe step="split" port="not-matched"/>
  </p:input>
</p:pack>

<p:wrap-sequence wrapper="sequence"/>
```

Figure 9. A simple XPROC pipeline

describe pipelines, which consist of heterogeneous steps, each performing some type of tree manipulation. The XPROC language support steps, which call programs written in other languages such as XSLT, XQUERY, or do schema validations using SCHEMATRON. The language also contains built-in steps, which perform simple operations such as element addition or removal as well as means to define complex steps by combining simple ones. The language contains limited support for branching and loops.

A simple XPROC pipeline is given in Figure 9. The first step of the pipeline takes an input data tree, extracts all subtrees rooted at a nodes, and forms a *sequence* of new data trees. This sequence is then passed to the `split-sequence` step, which splits the trees into two sequences. The split criterion is the XPATH expression `//b`, i.e., the presence of b nodes in the tree.

The next step in the pipeline, `pack`, takes two sequences and *zips* them together forming again a single sequence. This step has two input ports, which receive the trees. While the first input port is automatically linked to the first output port of the previous step, we need to link the second port explicitly. The `wrapper` argument provides a name for the element which will hold the two zipped trees.

The final step in the pipeline, takes a sequence of data trees and concatenates them together, forming a single tree. The input is automatically taken from the previous step, and the output of this step becomes the output of the whole pipeline.

***Compilation from* XPROC** X-Fun, with its external function interface can easily subsume XPROC. Complex XPROC steps, such as schema validation and XSLT transformation, or steps performing interactions with the environment, such as steps performing `http` requests, are translated as calls to external functions. The simple XPROC steps can be implemented as X-Fun functions. Figure 10 shows the translation of two of the simple XPROC steps mentioned above: `split-sequence`, `pack`.

```
split = fun (seq, test) : [node]×(node->[node])
        ->[node]×[node] {
  match seq {
    h|t -> let
      (matched, notmatched) = split(t, test) in
      if test(h) ≠ nil_node
        then (h|matched, notmatched)
        else (matched, h|notmatched)
    nil -> (nil_node, nil_node)
    }
  }

pack = fun (first, second, wrapper):
       [node]×[node]×label->[node] {
  match first*second {
    h|t ->
      let (fh, ft) = match first {
        h|t -> (map_node→tree(subtree,
                  evalPath_[node]("$h/node()")),
                t)
        nil -> (nil_tree, nil_node)
      } in
      let (sh, st) = match second {
        h|t -> (map_node→tree(subtree,
                  evalPath_[node]("$h/node()")),
                t)
        nil -> (nil_tree, nil_node)
      } in
      root_tree(makeTree(wrapper, fh*sh))
        | pack(ft, st, wrapper)
    nil -> nil_node
    }
  }
```

Figure 10. Compilation of XPROC steps

The rest of the XPROC constructs, such as choosing among many alternative sub-pipelines (`p:choose`), or looping over documents in a sequence (`p:for-each`) can be easily mapped to conditional expressions and recursion in X-Fun.

5. X-Fun as a core language

In this section we shall sketch translations from the most common XML processing languages, XSLT and XQUERY, into X-Fun. Since the specifications of the languages are quite voluminous and lack formal semantics, it is impossible to give precise translation algorithms with correctness proofs here. Even though there have been various formal models of these languages (like XQ_H [2] for XQUERY) in the literature, these models abstract from many of the features of the languages (for example, XQ_H lacks support for sorting, user-defined functions, etc.). Since our intention is to support a wider set of the language features than these models capture, our translations were implemented in practice but not formalized rigorously.

5.1 XSLT

We first recall the main concepts of XSLT and then show how to translate them to X-Fun. The supported fragment of XSLT includes named templates and template rules, both call instructions, conditional and iterative instructions, and dynamic content creation instructions (`element`, `value-of`, ...), grouping and sorting.

***Concepts of* XSLT** We demonstrate some of the concepts at the example of an XSLT program in Figure 11. An XSLT program consists of a set of templates, i.e., rules that can be applied to some node of an input data tree. The output produced by the template application is a combination of static data trees, data trees transformed form the inputs, and intermediate trees produced by recursive calls. There are two types of calls in the XSLT language: the `call-template` instruction calls a template by directly specifying its name, while `apply-templates` dynamically chooses a template satisfying the `match` filters and belonging to the specified `mode`. Ambiguities are solved by assigning explicit priorities to individual template rules. When using `apply-templates`, one can also navigate through the input tree as specified by the `select` query.

Both types of calls allow passing additional values as parameters, which are like global variables bound to some value.

The XSLT language supports the usual set of instructions for conditional and iterated evaluation like `if`, `choice`, `for-each`, etc. For specifying the conditions and selecting nodes to iterate over, as well as template match patterns one uses XPATH expressions. The translation starts in the *default mode* with the template matching the root node of the input tree.

Compiler to X-Fun A complete XSLT program is translated into an X-Fun program with a top-level let binder introducing all function definitions, followed by an application of the default mode function to the root node of the input. Each template with n parameters is translated to a function definition with $n+1$ arguments. The first argument `cur`, which passes the "current node" is of type **node**, while the others n arguments corresponding to parameters are of type [**node**] or **string**.

Furthermore, for each mode, we specify one additional function which selects the correct rule via a sequence of **if**-expressions and calls the corresponding function. Priorities of the rules are implemented using a suitable ordering of the tests clauses.

As an example of showing the translation technique, we consider the two XSLT templates in Figure 11, which are translated to the X-Fun definitions in Figure 12. This example shows how to translate template rules, both types of calls as well as parameters specified using select expressions and parameters with values constructed in-place. Template match patterns are rewritten into path expressions according to the rules in the XSLT specification [13]. The example also applies the built-in template rule of XSLT, which is applied if no rule matches the node.

```
<xsl:template name="p" match="a">
  <xsl:param name="y"/>
  <B>
    <xsl:apply-templates select="child::*">
      <xsl:with-param name="y" select="child::a"/>
    </xsl:apply-templates>
  </B>
</xsl:template>

<xsl:template name="q" match="b/c">
  <xsl:param name="y"/>
  <C>
    <xsl:call-template name="p">
      <xsl:with-param name="y">
        <A/>
      </xsl:with-param>
    </xsl:call-template>
    <xsl:copy-of select="$y"/>
  </C>
</xsl:template>
```

Figure 11. Technical example of XSLT templates, whose translation to X-Fun definitions is given in Figure 12.

```
p =
  fun (cur, y): node×[node]->hedge {
   [element(
     "B", let
       y'=evalPath[node]("$cur/child::a")
     in
       mapc node→hedge {(
        fun x':node->hedge {defaultMode(x', y')},
        evalPath[node]("$cur/child::*"))
       )]
   }
q =
  fun (cur, y): node×[node]->hedge {let
    y' = root tree(element("A", nil))
  in
    [element("C", p(cur, [y']) * map(subtree, y))]
  }
defaultMode =
  fun (cur, y) : (node×[node]) -> hedge {
    if member node(cur,
      evalPath[node]("root($cur)//a"))
    then p(cur, y)
    else if member node(cur,
      evalPath[node]("root($cur)//b/c"))
    then q(cur, y)
    else
      mapc (node×[node])->hedge(defaultMode,
           evalPath[node]("$cur/node()"))
  }
```

Figure 12. Translation from XSLT templates to X-Fun definitions.

Translation of other XSLT concepts is straightforward, therefore we only give a short overview in Figure 13. Sorting in `apply-templates` and `for-each` is also supported and translates to applying a sort operation on the result of the corresponding $\text{evalPath}_{[\text{node}]}$ call. The comparison function is implemented by computing each sorting key with $\text{evalPath}_{\text{string}}$ and then comparing the returned values. Grouping by `for-each-group` is implemented in terms of the X-Fun node grouping function. The result of evaluating the text-returning XPATH is the grouping key.

Limitations The most important limitation of this translation technique is the typing of the XPATH expressions. Basically, the expressions are typed by the context where they appear in. E.g., the `select` expression in `apply-templates` is of type $[\mathbf{node}]$, the `test` in the `if` element returns **bool**, etc. While this covers the common cases, it does not handle situations, where one calls `apply-templates` on a sequence of numbers, or XPATH sequences containing a mixture of nodes and strings. Formal approaches to studying the properties of XPATH [2, 5, 11] usually limit themselves to the navigational fragment of the language (while possibly allowing expressions of different types in filters), but for tree transformation the non-navigational aspects become relevant as well. Therefore, we chose not to exclude these features completely, but rather to support the widest fragments possible. Similar typing restrictions shall also apply to the XQUERY compiler.

5.2 XQUERY

XQUERY is a functional language for querying XML data trees, while outputting the results of the query as a new XML tree. Unlike XSLT and XPROC, the syntax of XQUERY is not XML-based. XQUERY also supports navigation using XPATH expressions, functions, control-flow expressions, and a SQL-like FLWOR (**for**, **let**, **where**, **order by**, **return**) expression.

XQUERY overview The feature that most distinguishes XQUERY from other functional languages is the FLWOR expression. It enables the the programmer to create a stream of tuples using the **for** and **let** clauses, filter them with a **where** clause, and then reorder them with **order by**. Below we show a FLWOR expression using most of these features:

```
for $a in $root//a
  let $b = $a/b
    where string($a/c) != 'hiho'
    order by string($b) ascending
    return <c> { $a } </c>
```

This expression first generates a list of tuples $(\$a, \$b)$, where $\$a$ is an a-node in the input tree and $\$b$ is the list of its b-labeled children. This list is then filtered according to the string value of the c child of $\$a$, and then sorted according to the string value of $\$b$. The return value of the expression is a hedge consisting of c element nodes, with a copies of $\$a$ as their children.

The other important feature of XQUERY is a *quantified* expression, most commonly used in a **where** clause of a FLWOR expression. The syntax of an existential quantified expression is:

```
some $x in E1 satisfies E2
```

Here, $E1$ is evaluated to produce a list of nodes, and the Boolean expression $E2$ is evaluated with $\$x$ bound to each node in turn. The result is the disjunction of these values. Universal semantics can be obtained by replacing `some` with `every`.

Compilation to X-Fun While in the case of XSLT, the separation between the navigational (XPATH) part and the tree-generation (XSLT) was very strict, in XQUERY they are more organically interwoven. Nevertheless, XQUERY can be split it the same way, and we shall translate it using the same strategy as XSLT: translate the tree-generation to X-Fun and leave the XPATH bits as is.

```
let
  list = map_{node->(node×[node])}(
           fun a:node -> node×[node] {
             (a, evalPath_{[node]}("$a/b"))
           },
           evalPath_{[node]}("$root//a")) in let
  filtered =
    filter_{node×[node]}(
      fun (a, b): node×[node] -> bool {
        evalPath_{bool}("string($a/c) != 'hiho'")
        },
      list) in let
  sorted =
    sort(filtered,
      fun (x,y): (node×[node])×(node×[node])->
          bool { let
        (xa, xb) = x in let
        (ya, yb) = y
      in
        evalPath_{bool}("string($xa) < string($xb)")
      })
in
  mapc_{node→[tree]}(
    fun (a,b): (node×node) -> [tree] {
      element("c", [subtree(a)])
      },
    sorted)
```

Figure 14. A translation of a small FLWOR expression

The translation of FLWOR expressions follows the semantics, as outlined above: we first generate the tuples, then filter, sort, and finally transform them. The XQUERY program from the beginning of the section is then translated to program in Figure 14.

Quantified expressions are translated to a function, which processes the list returned by E_1 recursively, applies (translation of) E_2 to each item in the list, and computes the conjunction/disjunction of the results. Other XQUERY constructs, like the **if** expression and node construction expressions map to equivalent constructs in X-Fun: the conditional expression and makeTree calls. The functions of XQUERY are translated to X-Fun functions in a straight forward way. We support functions taking (lists of) nodes or strings as parameters.

5.3 XPROC

In Section 4 we showed how to translate XPROC pipelines into X-Fun programs using XSLT, XQUERY, etc. implementations as external functions. In this section we have demonstrated that these transformations can also be expressed within X-Fun. Therefore, we can go a step further and replace external function calls with calls to X-Fun functions, obtained by translating the respective programs into X-Fun. By doing this we obtain a single core language that can express all stages of XML processing.

6. Implementation and Experiments

We have implemented the X-Fun language evaluator in the Java programming language. We have instantiated X-Fun with the XML data model, using standard Java libraries for manipulating XML trees. We have used XPATH as the path language, also using the standard Java interface, as implemented by SAXON. We have used standard techniques for implementing functional languages, using the heap to store the values and the environment of the program and a stack for representing recursive function calls. We reduce

Figure 13. XSLT instructions and their X-Fun equivalents

an expression in all possible positions in an arbitrary order. When we get blocked, since the value of some variable is needed from the environment, but not yet bound there, we suspend the current activity. It will be reactivate it later on once the variable gets bound (if any). If the evaluation reaches a value, we are done. If the evaluation stops, while some activities remains, an error is raised. Errors may also be raised for the other reasons mentioned earlier.

Although all the features of the language are implemented, the implementation should only be considered as a proof-of-concept so far, as we have not spent time optimising the time and memory usage. We have also interfaced our implementation with TATOO, an efficient evaluator of an XPATH fragment based on [1]. Unfortunately, the penalty of crossing the language barrier (TATOO is implemented in OCAML) shadowed all performance gains from a faster implementation, so we could not perform any significant experiments. To see the difference in performance in using a faster XPATH implementation, one would need to implement X-Fun in OCAML as well.

We have also implemented the compilers of XSLT and XQUERY into X-Fun, which operate by applying the rewrite rules laid out in Section 5. In order to support real-world XSLT and XQUERY, they need support for additional features, like modules and various optional attributes of expressions in these languages (e.g., grouping with the `group-starting-with` attribute, etc.). However, none of these limitations (apart from XPATH typing described earlier) are fundamental and they are not implemented because of their volume.

We don't have an XPROC compiler implementation, but for the purposes of testing we have run X-Fun on programs manually translated using the rules outlined in Section 4.

6.1 Experiments

To evaluate the performance of our implementation, we have compared it with the leading industry tool, the SAXON XSLT and XQUERY processor. To compare our performance on XPROC pipelines, we have used CALABASH, the most frequently used XPROC processor, as baseline. The tests were run on a computer with an Intel Core i7 processor running at 2.8 GHz, with 4GB or RAM and a SATA hard drive, running 64-bit Linux operating system.

Query	X-Fun	SAXON
Q1	30 s	8.7 s
Q6	29 s	9.1 s
Q13	12 s	9.4 s
Q15	90 s	9.6 s
Q17	14 s	9.5 s
Q20	24 s	11.2 s

Figure 15. Performance of X-Fun and SAXON on queries from the XMARK benchmark on a 100 MB document

We have also compared the running time of our implementation on XQUERY programs. We ran several queries from the XMARK benchmark, and the results are in Figure 15. The tests show that SAXON is faster by a factor of 2–9 than our tool and that the time of both tools scales linearly with the document size. Additionally, we have run the query $Q8$ on a 20 MB documents. The times were 4.4 vs. 56 seconds in favor of SAXON, but now both tools showed quadratic dependence on document size, which was expected, since the query contains joins. The difference in performance does not come as a surprise, since SAXON is a highly optimised industry tool, while we have not spent much time optimising the performance of our X-Fun implementation. We believe that with further implementation optimisations we could close the performance gap.

For the XSLT test, we used a transformation publishing an address book to HTML. The transformation in question is a more elaborate version of the program in Figure 7, and it includes about 40 XPATH expressions. The tests show that SAXON is about 6 times faster than our tool (for example, 5.6 vs. 33 seconds on a 30 MB document) and that the time of our tool scales linearly with the document size. These results are consistent with the XQUERY benchmark.

In the XPROC comparison, we have used the pipeline from the example in Section 5.3, and compared the performance of

Document size	X-Fun	CALABASH
1 MB	5.3 s	11 s
2 MB	7.6 s	31 s
3 MB	10.3 s	54 s
4 MB	12.4 s	75 s

Figure 16. Performance of X-Fun and CALABASH on a fixed pipeline with varying input tree size

Pipeline size	X-Fun	CALABASH
1	5.3 s	11 s
2	10 s	31 s
3	14 s	54 s
4	18 s	75 s

Figure 17. Performance of X-Fun and CALABASH on a 1 MB document with varying pipeline size

CALABASH with our implementation of the pipeline in X-Fun. Both implementations show linear scalability with respect to size of the input and the pipeline, as can be seen in Figures 17 and 9 (for scaling the pipeline size, we simply composed the pipeline from Section 5.3 with itself). However, our own implementation is consistently about 2–5 times faster. While the relatively low processing speed per megabyte can be explained by the need to create many small documents (the element per megabyte density is much higher compared to the previous tests), it is surprising to see an implementation specifically designed for processing XPROC be outperformed by our unoptimised implementation of the pipeline steps.

7. Conclusion and future work.

We have presented X-Fun, a language for processing data trees, and shown that can serve as a universal pipeline language for XML processing and as a uniform core language for implementing XQUERY, XSLT, and XPROC on top of any existing XPATH evaluator. Our implementation based on SAXON's in-memory XPATH evaluator yields surprisingly efficient implementations of the three W3C standards, even without optimization, which is left to future work. We are optimistic that one can obtain equally good results as with SAXON's XQUERY and XSLT evaluators, but this remains open so far. In the case of XPROC, first results show that we are already faster than CALABASH without any optimization.

Our prime objective in future work is to build streaming implementations of X-Fun, and thus XQUERY, XSLT, and XPROC. The main ideas behind are described in a technical report [16]. These streaming implementation will serve in the tools called QUIXQUERY, QUIXSLT, and QUIXPROC. A first version of QUIXSLT is freely available for testing on our online demo machine [22] while streaming is not yet available for our current QUIXPROC implementation.

Acknowledgement

We would like to thank Guillaume Bagan for his help with implementing the XQUERY compiler and Denis Debarbieux for work on the foreign function interface of X-Fun. Thanks also goes to Tom Sebastian and Mohamed Zergaoui for their comments during our discussions.

References

[1] D. Arroyuelo, F. Claude, S. Maneth, V. Mäkinen, G. Navarro, K. Nguyen, J. Sirén, and N. Välimäki. Fast in-memory xpath search using compressed indexes. In *ICDE*, pages 417–428. IEEE, 2010.

[2] M. Benedikt and H. Vu. Higher-order functions and structured datatypes. In Z. G. Ives and Y. Velegrakis, editors, *15th International Workshop on the Web and Databases*, pages 43–48, 2012.

[3] V. Benzaken, G. Castagna, and A. Frisch. CDuce: an XML-centric general-purpose language. *ACM SIGPLAN Notices*, 38(9):51–63, 2003. URL `http://doi.acm.org/10.1145/944746.944711`.

[4] V. Benzaken, G. Castagna, K. Nguyen, and J. Siméon. Static and dynamic semantics of NoSQL languages. In R. Giacobazzi, R. Cousot, R. Giacobazzi, and R. Cousot, editors, *POPL*, pages 101–114. ACM, 2013. ISBN 978-1-4503-1832-7. doi: 10.1145/2429069.2429083. URL `http://dx.doi.org/10.1145/2429069.2429083`.

[5] G. Castagna, H. Im, K. Nguyn, and V. Benzaken. A Core Calculus for XQuery 3.0. Unpublished manuscript, 2013.

[6] Ecma International. The JSON Data Interchange Format, 2013. http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf.

[7] A. Frisch and K. Nakano. Streaming XML transformation using term rewriting. In *Programming Language Technologies for XML (PLAN-X 2007)*, 2007.

[8] Z. Fülöp and H. Vogler. *Syntax-Directed Semantics – Formal Models based on Tree Transducers*. EATCS Monographs in Theoretical Computer Science (W. Brauer, G. Rozenberg, A. Salomaa, eds.). Springer-Verlag, 1998.

[9] S. Goessner. JSONPath - XPath for JSON, Feb. 2007. http://goessner.net/articles/JsonPath/.

[10] G. Gottlob and C. Koch. Monadic queries over Tree-Structured data. In *17th Annual IEEE Symposium on Logic in Computer Science*, pages 189–202, Copenhagen, 2002.

[11] S. Hakuta, S. Maneth, K. Nakano, and H. Iwasaki. XQuery Streaming by Forest Transducers. In *30th IEEE International Conference on Data Engineering*. IEEE Computer Society, 2014.

[12] Innovimax. QuiXProc. http://www.quixproc.com.

[13] M. Kay. *XSL Transformations (XSLT) Version 3.0*. W3C Last Call Working Draft, 2013. http://www.w3.org/TR/xslt-30.

[14] S. Kepser. A simple proof for the Turing-Completeness of XSLT and XQuery. In *Extreme Markup Languages®*, 2004. URL `http://dblp.uni-trier.de/rec/bibtex/conf/extreme/Kepser04`.

[15] C. Kirchner, P. E. Moreau, and C. Tavares. A type system for tom. In I. Mackie, A. M. Moreira, I. Mackie, and A. M. Moreira, editors, *RULE*, volume 21 of *EPTCS*, pages 51–63, 2009. doi: 10.4204/eptcs.21.5. URL `http://dx.doi.org/10.4204/eptcs.21.5`.

[16] P. Labath and J. Niehren. A Functional Language for Hyperstreaming XSLT. Research report, Mar. 2013. URL `http://hal.inria.fr/hal-00806343`.

[17] S. Maneth, A. Berlea, T. Perst, and H. Seidl. XML type checking with macro tree transducers. In *24th ACM Symposium on Principles of Database Systems*, pages 283–294, New York, NY, USA, 2005. ACM-Press. URL `http://doi.acm.org/10.1145/1065167.1065203`.

[18] R. Milner. A proposal for standard ml. In *Symposium on LISP and Functional Programming*, pages 184–197, New York, NY, USA, 1984. ACM. ISBN 0-89791-142-3.

[19] K. Nakano and S.-C. Mu. A Pushdown Machine for Recursive XML Processing. In N. Kobayashi, editor, *Programming Languages and Systems*, LNCS, pages 340–356. Springer Berlin Heidelberg, 2006.

[20] A. Neumann and H. Seidl. Locating matches of tree patterns in forests. In *18-th Conference on Foundations of Software Technology and Theoretical Computer Science*, 1998. doi: 10.1007/b71635. URL `http://dx.doi.org/10.1007/b71635`.

[21] S. L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, 1987. URL `http://dblp.uni-trier.de/rec/bibtex/books/ph/Jones87`.

[22] QuiX tools suite. URL `https://project.inria.fr/quix-tool-suite/`.

[23] J. Robie, D. Chamberlin, M. Dyck, and J. Snelson. *XML Path Language (XPath) 3.0*. W3C Proposed Recommendation, 2013. http://www.w3.org/TR/xpath-30.

[24] J. Robie, D. Chamberlin, M. Dyck, and J. Snelson. *XQuery 3.0: An XML Query Language*. W3C Proposed Recommendation, 2013. http://www.w3.org/TR/xquery-30.

[25] Saxonica. SAXON 9.5: The XSLT and XQuery Processor. http://saxonica.com.

[26] G. Smolka. Objects in a higher-order concurrent constraint model with state. In A. Napoli and A. Napoli, editors, *LMO*, pages 69–74. INRIA, 1995. URL `http://dblp.uni-trier.de/rec/bibtex/conf/lmo/Smolka95`.

[27] N. Walsh. XML Calabash. http://xmlcalabash.com.

[28] N. Walsh, A. Milowski, and H. S. Thompson. XProc: An XML Pipeline Language, Mar. 2009.

Appendix

Here we provide additional examples of X-Fun programs. In Figure 18 we provide an implementation of the group function in X-Fun, while Figure 19 contains an implementation of an additional XPROC step.

```
fun (l,q):[node]×(node->string) -> [[node]] {let
  # groups a list of nodes l according to
  # the string selected by a query q
  l2 = map_{node->(string×node)}(
         fun n:node -> string×node {
           (q(n), n)
           },
         l),
  split =
    fun (k,l):string×[string×node]
        -> [node]×[node]  {
      # Receives a string k and a list l.
      # Splits list l into pairs (k2,l)
      # satisfying k=k2 and the rest.
      match l {
        h|t -> let
          (k2, n) = h in let
          (g, rest) = split(k, t)
        in
          if k = k2 then (n|g, rest)
          else (g, n|rest)
        nil -> (nil_{node}, nil_{node})
        }
    },
  groupByString =
    fun l: [string×node]->[[node]] {
      # Splits lists l of pairs (k,n) into
      # groups g of nodes n having the same
      # string k in the first component
      match l {
         h|t -> let
           (k, n) = h in let
           (g, rest) = split(k, l)
         in
           g | groupByString(rest))
         nil -> nil_{[node]}
         }
      }
in
  groupByString(l2)
}
```

Figure 18. Group function in X-Fun.

```
delete =
  fun (doc, select): node×(node->[node])
       ->node { let
    set = evalPath_{[node]}(select(doc)) in let
    rec =
      fun x: node->hedge {
        if member_{node}(x, set)
        then nil_{hedge}
        else [makeTree(label(x),
             mapc_{node→[tree]}(rec,
                  evalPath_{[node]}("$x/node()"))
          )]
        }
  in
    root_{tree}(rec(doc))
  }
```

Figure 19. The implementation of the XPROC `delete` step.