A Uniform Programming Language for Implementing XML Standards*

Pavel Labath[1] and Joachim Niehren[2]

[1] Commenius University, Bratislava [2] INRIA, Lille

Abstract. We propose X-Fun, a core language for implementing various Xml standards in a uniform manner. X-Fun is a higher-order functional programming language for transforming data trees based on node selection queries. It can support the Xml data model and XPath queries as a special case. We present a lean operational semantics of X-Fun based on a typed lambda calculus that enables its in-memory implementation on top of any chosen path query evaluator. We also discuss compilers from Xslt, XQuery and XProc into X-Fun which cover the many details of these standardized languages. As a result, we obtain in-memory implementations of all these Xml standards with large coverage and high efficiency in a uniform manner from Saxon's XPath implementation.

Keywords: Xml transformations, database queries, functional programming languages, compilers.

1 Introduction

A major drawback of query-based functional languages with data trees so far is that they either have low coverage in theory and practice or no lean operational semantics. Theory driven languages are often based on some kind of macro tree transducers [12,5,3], which have low coverage, in that they are not closed under function composition [4] and thus not Turing complete (for instance type checking is decidable [11]). The W3C standardised languages XQuery [13] and Xslt [7], in contrast, have large coverage in practice (string operations, data joins, arithmetics, aggregation, etc.) and in theory, since they are closed by function composition and indeed Turing complete [8]. The definitions of these standards, however, consist of hundreds of pages of informal descriptions. They neither explain how to a build a compiler in a principled manner nor can they be used as a basis for formal analysis.

A second drawback is the tower of languages approach, adopted for standardised Xml processing languages. 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

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

queried. Since the combination of both is needed in most larger applications, the XML pipeline language XPROC [17,16,18] was developed and standardised again by the W3C. This resulted in yet another functional programming language for processing data trees based on XPATH.

For resolving the above two drawbacks, the question is whether there exists a uniform core language for processing data trees that can cover the different XML standards in a principled manner. It should have a lean and formal operational semantics, support node selection queries as with XPATH and it should be sufficiently expressive in order to serve as a core language for implementing XQUERY, XSLT, and XPROC in a uniform manner.

Related work. An indicator for the existence of a uniform core language for XML processing is that the omnipresent Saxon system [14] implements XSLT and XQUERY on a common platform. However, there is no formal description of this platform as a programming language, and it does not support the XML pipeline language XPROC so far. Instead, the existing implementations of XPROC, CALABASH [16] and QUIXPROC [18], are based on Saxon's XPATH engine directly.

The recent work from Castagna et. al. [2] 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, which 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). Therefore, our last requirement is not satisfied.

Contributions. In this paper, we present the first positive answer to the above question based on X-Fun. This is a new purely functional programming language. X-Fun is a higher-order language and it supports the evaluation of path-based queries that select nodes in data trees. The path queries are mapped to X-Fun expressions, whose values can be computed dynamically. In contrast to most previous interfaces between databases and programming languages, we overload variables of path queries with variables of X-Fun. In this manner, the variables in path queries are always bound to tree nodes, before the path query is evaluated itself. We note in particular, that path queries are not simply mapped to X-Fun expressions of type string.

The formal model of the operational semantics of X-Fun is a lambda calculus with a parallel call-by-value reduction strategy. Parallel evaluation is possible due to the absence of imperative data structures. The main novelty in X-Fun admission of tree nodes as values of type **node**. Which precise nodes are admitted depends on a tree store. New nodes can be created dynamically by adding new trees to the tree store. The same tree can be added twice to the store but with different nodes. How nodes are represented internally can be freely chosen by the X-Fun implementation and is hidden from the programmer.

X-Fun can serve as a uniform core language for implementing XQuery, Xslt and XProc. In order to do so, we have developed 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. Based on our compilers, we thus obtain new in-memory implementations of XQuery, Xslt and XProc with large coverage. Our implementation has very good efficiency and outperforms the most widely used XProc implementation by a wide margin.

Outline. In Section 2 we introduce our general model of data trees, alongside its application to Xml documents. The syntax and type system of the X-Fun language is introduced in Section 3. The applications of X-Fun to Xml document transformation is studied in Section 4, where we discuss compilers from other Xml processing languages into X-Fun. Section 5 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 which will be used in the X-Fun language. We also show how to instantiate the trees to the Xml data model.

2.1 Data values and data trees

We fix a finite set *Char* whose elements will be called characters. A data value "$c_1 \cdots c_n$" is a word of characters for $c_1, \ldots, c_n \in \mathit{Char}$. We define $\mathit{String} = \mathit{Char}^*$ 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 $(\mathit{String})^k$. 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 abstract syntax $t ::= \; l(t_1, \ldots, t_n)$, where $n \geq 0$ and $l \in \mathit{String}^k$. It should be noticed that the set of node labels is infinite, but that each node label can be represented finitely.

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", nil, nil, v_3)	("processing-instruction", v_1, nil, v_3)
("document", nil, nil, nil)	("text", v_1, nil, nil)

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

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.

We start with introducing the types and values of X-Fun (Section 3.1). Then we explain how to map path queries to X-Fun values, by using particular X-Fun expressions with variables (Section 3.2). The general syntax of X-Fun expressions is given in Section 3.3. Some syntactic sugar and an example of an X-Fun program are given in Sections 3.7 and 3.8. Discussion of the typing rules for X-Fun's type system and the formal semantics of X-Fun can be found in the research report [10].

3.1 Types and Values

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

$$\begin{aligned} T ::= &\ \mathbf{none} \ \mid\ \mathbf{node} \ \mid\ \mathbf{tree} \ \mid\ \mathbf{number} \ \mid\ \mathbf{bool} \ \mid\ \mathbf{char} \\ &\mid\ T_1 \times \ldots \times T_n \ \mid\ [T] \ \mid\ T_1 \to T_2 \ \mid\ T_1 \cup T_2 \end{aligned}$$

A value of type **char** is an element of *Char*, a value of type **tree** is an element of $\mathcal{T}$. A value of type **number** is a floating point number, while the values of type **bool** are the Boolean values *true* and *false*. A value of type **node** will be a node of the graph of one of the trees stored by the environment of the X-Fun evaluator. The precise node identifiers chosen by the evaluator are left internal (to the mapping from trees to graphs).

As usual, we support list types $[T]$ which denote 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. Besides these, we also support type unions in the obvious manner.

A data value $"c_1 \cdots c_n" \in \mathit{String}$ is considered as a list of characters of type **string** $= [$**char**$]$. A node label is considered a k-tuple of strings, i.e., as a value of type **label** $=$ $\mathbf{string}^k$. Hedges are considered as lists of trees of type **hedge** $= [$**tree**$]$.

Since XPATH can return sequences of items of different types, we define the type **pathresult** as **node** $\cup$ **number** $\cup$ **string** $\cup$ **bool**. The result of evaluating a path expression will then be of type $[$**pathresult**$]$. To be able to specify path expressions, we define the type **path** as $[$**char** $\cup$ **pathresult** $\cup$ $[$**pathresult**$]]$, i.e., as list of characters, individual items returned by a path expression, and whole sequences of those items.

3.2 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]
```

then one might want to evaluate this expression while variable x is bound to a node of some tree and variable y to some data value. In X-Fun, the above query will be represented by the following expression of type **path**, where x is a variable of type **node** and y a variable of type **string**:

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

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

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

In order to enable the evaluation of path expressions, X-Fun supports a builtin function evalPath of type **path** $\to$ [**pathresult**]. In an implementation of X-Fun, this function can be mapped straightforwardly to existing XPATH evaluators.

3.3 Syntax of X-Fun expressions

X-Fun is a purely functional programming language whose values subsume higher-order function, trees, strings, numbers and Boolean values. 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, so we only briefly describe different kinds of subexpressions of X-Fun programs.

A variable x is evaluated to the value of the corresponding type. The constant expression c returns the respective constant, which can be a Boolean value, a number or a character from *Char*. The list constructor $E_1 :: E_2$ prepends an element to a list, while the tuple constructor $(E_1, \ldots, E_n)$ constructs tuples.

The match expression **match** E $\{ P_1 \to E_1, \ \ldots, \ P_n \to E_n \}$ selects one of the branches E_i based on the patterns P_i, which are matched against the value of E. The pattern $x : T$ captures a matched value of type T into a variable. The pattern $!(E)$ matches the value against the value of expression E. Here, the matching of functional values, or lists/tuples that contain functions is not permitted. Pattern $P_1 :: P_2$ matches a list if P_1 and P_2 match its head and tail, while the pattern $(P_1, \ldots, P_n)$ matches tuples.

A function expression **fun** $x{:}T_1 \to T_2$ $\{ E \}$ returns a new function, with the argument $x : T_1$ and the 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. X-Fun also supports exception handling, where exceptions are values of type **string**.

Expressions

$$
\begin{array}{rl}
E ::= & x \\
\mid & c \\
\mid & E_1 :: E_2 \\
\mid & (E_1, \ldots, E_n),\ n \geq 2 \\
\mid & \textbf{match}\ E\ \{\ P_1 \rightarrow E_1,\ \ \ldots,\ P_n \rightarrow E_n\ \} \\
\mid & \textbf{fun}\ x{:}T_1 \rightarrow T_2\ \{\ E\ \} \\
\mid & E_1(E_2) \\
\mid & \textbf{try}\ E_1\ \textbf{catch}(x)\ E_2 \\
\mid & \textbf{raise}(E)
\end{array}
$$

Patterns

$$
\begin{array}{rl}
P ::= & x : T \\
\mid & !(E) \\
\mid & P_1 :: P_2 \\
\mid & (P_1, \ldots, P_n),\ n \geq 2
\end{array}
$$

Fig. 1. Syntax of X-Fun's expressions

3.4 Typing

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 typing rules of X-Fun are explained in Figure 2. Most of them are standard. However, it should be noticed that we require explicit type annotations on function definitions, since this type cannot always be inferred otherwise. In the figure, the relation $T_1 \preceq T_2$ means that the type T_2 is more general than T_1, i.e., that set of values of type T_1 is a subset of the set of values of type T_2. The definition of this subtyping relation is given in Figure 3. These rules are also self-explaining so we do not go into detail. We only single out the last two rules on tuples, which allow us to deduce that types $(T_1, T_2 \cup T_2')$ and $(T_1, T_2) \cup (T_1, T_2')$ are equivalent

A complete program is an expression of function type $T_1 \rightarrow T_2$, for which all free variables are bound by the original environment, such as those in Figure 5. The arguments of the function will be provided when the function is called.

3.5 Semantics

We define the semantics of X-Fun by the small-step evaluator in Figure 4, which rewrites expressions to expressions 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 added 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 com-

Expressions $\dfrac{E : T \qquad T \preceq T'}{E : T'}$ $\quad \dfrac{\textsc{Type}(x) = T}{x : T}$ $\quad \dfrac{c \in \mathit{Char}}{c : \textbf{char}}$ $\quad \dfrac{n \in \mathbb{R}}{c : \textbf{number}}$

$\dfrac{b \in \{\mathit{true}, \mathit{false}\}}{c : \textbf{bool}}$ $\quad \dfrac{E_1 : T \qquad E_2 : [T]}{E_1 :: E_2 : [T]}$ $\quad \dfrac{E_i : T_i}{(E_1, \ldots, E_n) : T_1 \times \cdots \times T_n}$

$\dfrac{E : T \qquad P_i : T \qquad E_i : T'}{\textbf{match}\ E\ \{\ P_1 \to E_1,\ \ldots,\ P_n \to E_n\ \} : T'}$ $\quad \dfrac{E_1 : T \to T' \qquad E_2 : T}{E_1(E_2) : T'}$

$\dfrac{\textsc{Type}(x) = T_1 \qquad E : T_2}{\textbf{fun}\ x{:}T_1 \to T_2\ \{\ E\ \} : T_1 \to T_2}$ $\quad \dfrac{E_1 : T \qquad \textsc{Type}(x) = \textbf{string} \qquad E_2 : T}{\textbf{try}\ E_1\ \textbf{catch}(x)\ E_2 : T}$

$\dfrac{E : \textbf{string}}{\textbf{raise}(E) : \textbf{none}}$

Patterns $\dfrac{P : T \qquad T \preceq T'}{P : T'}$ $\quad \dfrac{\textsc{Type}(x) = T}{[x : T] : T}$ $\quad \dfrac{P_1 : T \qquad P_2 : [T]}{P_1 :: P_2 : [T]}$

$\dfrac{E : T \qquad T \text{ non-functional}}{!(E) : T}$ $\quad \dfrac{P_i : T_i}{(P_1, \ldots, P_n) : T_1 \times \cdots \times T_n}$

Fig. 2. Typing rules of X-Fun.

$\dfrac{\mathit{true}}{\textbf{none} \preceq T}$ $\quad \dfrac{T_1 \preceq T_2}{[T_1] \preceq [T_2]}$ $\quad \dfrac{T_1' \preceq T_1 \qquad T_2 \ \preceq\ T_2'}{T_1 \to T_2 \preceq T_1' \to T_2'}$ $\quad \dfrac{\mathit{true}}{T \preceq T \cup T'}$

$\dfrac{T_1 \preceq T \qquad T_2 \preceq T}{T_1 \cup T_2 \preceq T}$ $\quad \dfrac{T_1 \preceq T_2}{T_1 \cup T_2 \preceq T_2}$ $\quad \dfrac{T_i \preceq T_i'}{T_1 \times \cdots \times T_n \preceq T_1' \times \cdots \times T_n'}$

$$\dfrac{T_i \preceq T_i' \cup T_i''}{(T_1 \times \cdots \times T_n) \preceq (T_1 \times \cdots \times T_i' \times \cdots \times T_n) \cup (T_1 \times \cdots \times T_i'' \times \ldots \times T_n)}$$

Fig. 3. The subtyping relation of X-Fun.

Evaluation $$\frac{E \xrightarrow{\eta} E' \qquad C[.] \text{ evaluation context} \qquad \eta \text{ action on D}}{C[E] \xrightarrow{\eta} C[E']}$$

$$\frac{v = \mathit{getVal}^D(x)}{x \to v} \qquad \frac{i \text{ least integer s.t. } [\![v]\!]_{P_i} = F_i}{\textbf{match } v \ \{\ P_1 \to E_1, \ldots, P_n \to E_n\ \} \xrightarrow{\mathit{bindVars}_D(F_i)} E_i}$$

$$\frac{\mathit{true}}{\textbf{fun } x{:}T_1 \to T_2\ \{\ E\ \}(v) \to \textbf{match } v\ \{\ x' : T_1 \to E[x/x']\ \}} \qquad \frac{t = \mathit{subtree}_D(v)}{\mathrm{subtree}(v) \to t}$$

$$\frac{t \text{ is a tree}}{\mathrm{addTree}(t) \xrightarrow{v = \mathit{addTree}_D(t)} v} \qquad \frac{l(h) \text{ valid data tree of the chosen data model}}{\mathrm{makeTree}((l, h)) \to l(h)}$$

$$\frac{v = \mathit{evalPath}_D(w)}{\mathrm{evalPath}(w) \to v}$$

Pattern matching $$\frac{v \text{ is a value of type } T}{[\![v]\!]_{x:T} = (x = v)} \qquad \frac{v \text{ is a value}}{[\![v]\!]_{!(v)} = ()}$$

$$\frac{v = (v_1, \ldots, v_n) \qquad [\![v_i]\!]_{P_i} = F_i}{[\![v]\!]_{(P_1, \ldots, P_n)} = F_1 \& \cdots \& F_n} \qquad \frac{v = v_1 :: v_2 \qquad [\![v_i]\!]_{P_i} = F_i}{[\![v]\!]_{P_1 :: P_2} = F_1 \& F_2}$$

Fig. 4. Small-step semantics of X-Fun expressions in an environment D.

pute 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 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 evalPath(p) cannot proceed if p is a value of type **path** but does not represent a well-formed path query.

The semantics is defined with respect to an environment D. It defines two kinds of judgements $E \xrightarrow{\eta} E'$ 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 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_i are variables, and v_i

are values with respect to D:

$\mathit{bindVars}_D((x_1 = v_1)\& \cdots \&(x_n = v_n))$	for $i = 1, \ldots, n$ bind variable x_i to value v_i
$v = \mathit{getVal}_D(x)$	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 judgements, where v and v' are values and t is a data tree:

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

The precise definition of $\mathit{evalPath}_D$ 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 $C[\cdot]$ is an X-Fun expression E that contains a single occurrence of a fresh variable "$\cdot$" 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 **fun** $x\!:\!T_1 \to T_2 \;\{\; E \;\}$, and not in the continuations E_i of a match expression **match** $E \;\{\; P_1 \to E_1, \;\; \ldots, \;\; P_n \to E_n \;\}$. We will write $C[E']$ for the expression obtained by substituting the unique occurrence of the hole marker $\cdot$ by E.

We now discuss the rules of the operational semantics in Figure 4. 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.

The rule for **match** expression states that the expression can be replaced by the first expression E_i whose pattern P_i matches the value v. During the replacement all capture variables in the pattern P_i are bound to the respective sub-values of v in the environment. The judgements for patterns define whether a pattern matches a value. If the judgement $[\![v]\!]_P = F_1\& \cdots \&F_n$ holds then the pattern P matches the value v and the expression $F_1\& \cdots \&F_n$ provides a list of captured bindings.

The reduction of function applications **fun** $x\!:\!T_1 \to T_2 \;\{\; E \;\}(v)$ creates a **match** expression **match** $v \;\{\; x' : T_1 \to E[x/x'] \;\}$ binding a fresh variable x' to the argument v, 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 addTree(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 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 evalPath(w) is reduced to the result of evaluating query w with respect to D, i.e. $evalPath_D(w)$, if it exists.

3.6 Builtin operators

At the beginning of the evaluation, the environment contains bindings of the global variables given in Figure 5.

Parameters		Fixed	
Global variable	TYPE	Global variable	TYPE
makeTree	**label** $\times$ [**tree**] $\rightarrow$ **tree**	nil	[**none**]
evalPath	**path** $\rightarrow$ [**pathresult**]	subtree	**node** $\rightarrow$ **tree**
less	**char** $\times$ **char** $\rightarrow$ **bool**	label	**node** $\rightarrow$ **label**
		addTree	**tree** $\rightarrow$ **node**

Fig. 5. Builtin operators of X-Fun

The first block contains three functions, whose semantics are parameters of the language, and depend on the query language and data model. For a label l and a sequence of trees h, the function application makeTree(l, h) returns the data tree $l(h)$, if $l(h)$ is a well-formed data tree (e.g., in the XML data model attributes cannot have children) and raises an exception otherwise. The function evalPath(p) evaluates a path expression p. Whenever p is not well-formed (e.g., with respect to the XPATH 3.0 specification) 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. We will also define functions evalPath$_T$, on top of evalPath, for T = [**node**], [**string**], etc. These functions verify (using a **match** expression with a typecase) that the result of the path call is of type T and raise an exception otherwise.

The next four operators are generic and do not depend on the specific kind of data trees. The variable nil refers to the empty list. A function application subtree(v) returns the subtree rooted at node v, while a function application label(v) returns the label of the node. The function addTree 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.

3.7 Syntactic sugar

In the X-Fun snippets in the rest of the paper we shall employ some syntactic shortcuts, which enable us to express more succinctly some X-Fun constructs:

list concatenation We shall use the binary operator `*` to concatenate two lists.

simplified patterns When the type of a capture variable can be deduced from the matched expression we shall omit the "$: T$" in the capture pattern. This happens when the **match** expression is used to decompose lists and tuples instead of doing a typecase. For example, we shall simply write **match** E { $h :: t \ \rightarrow\ E_1,\ e\ \rightarrow\ E_2$ } to get the head and tail of a list.

let-declarations We shall use the syntax **let** $x_1 = E_1, \ldots, x_n = E_n$ **in** E instead of **match** $(E_1, \ldots, E_n)$ { $(x_1, \ldots, x_n) \ \rightarrow\ E$ } as a more familiar way to declare variables.

tuple arguments We shall allow tuple arguments to functions to be written without an extra pair of parentheses. I.e., $f(a, b)$ instead of $f((a, b))$. This is unambiguous since tuples always have at least two members.

equality comparison The operator $E_1 = E_2$ shall be defined (for non-functional types T) as **match** E_1 { $!(E_2) \rightarrow \mathit{true},\ x : T \rightarrow \mathit{false}$ }.

conditional expression The expression **if** E **then** E_1 **else** E_2, where E is of type **bool** is defined as **match** E { $!(\mathit{true}) \rightarrow E_1,\ !(\mathit{false}) \rightarrow E_2$ }.

multi-argument functions For functions accepting tuples as arguments, we shall write the expression **fun** $(a, b) : T_1 \times T_2 \rightarrow T$ { E } instead of **fun** $x : T_1 \times T_2 \rightarrow T$ followed by a **match** x { $(a, b) \rightarrow E$ }

3.8 Example

In Figure 6 we illustrate a transformation that converts an address book into Html. 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` and, `phone`.

```
<addresses>                                  <ol>
<address>                                    <li>
 <name>Jemal Antidze</name>                    <p>Jemal Antidze</p>
 <phone>99532 305972</phone>                   <p>Tblissi</p>
 <city>Tblissi</city>                          <p>Phone: 99532 305972</p>
 <phone>99532 231231</phone>                   <p>Phone: 99532 231231</p>
</address>                                 ⇒ </li>
<address>                                    <li>
 <name>Joachim Niehren</name>                  <p>Joachim Niehren</p>
 <city>Lille</city>                            <p>Rue Esquermoise</p>
 <street>Rue Esquermoise</street>              <p>Lille</p>
</address>                                   </li>
</addresses>                                 </ol>
```

Fig. 6. Publication of an address book in Html except for secret entries

An X-Fun program defining this transformation is given in Figure 7. Starting at the root it 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 variables named alike and later output as text nodes. The example program uses the standard map function, which can be defined in X-Fun for every T and T' as follows

```
map_{T→T'} = fun x: (T → T') × [T] → [T'] { match x {
    (f, head::tail) → f(head)::map_{T→T'}(f, tail)
    other → nil
} }
```

and the functions element and text, which are wrappers around makeTree which facilitate creation of nodes of the correct kind.

```
fun book : tree→tree {
 let bookroot = addTree(book) in
 let convert_address = fun x : node→tree {
    let name = evalPath_[node]("$x/child::name/text()"),
          street = evalPath_[node]("$x/child::street/text()"),
          city = evalPath_[node]("$x/child::city/text()") in
    element("li",
       element("p", map_node→tree(name, subtree))::
       element("p", map_node→tree(street, subtree))::
       element("p", map_node→tree(city, subtree))::
       map_string→tree(
          fun x: string → tree {
             element("p", text("Phone: " * x)::nil)
          }, evalPath_[string]("data($x/child::phone)"))
    )
  } in
  element("ol", map_node→tree(convert_address,
       evalPath_[node]("$bookroot/descendant::address")))
}
```

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

4 Translations from other XML languages

In this section, we briefly sketch translations from the standard XML processing languages, XSLT XQUERY and XPROC. By implementing these three compilers, we obtain a uniform implementation of the whole XML processing stack based on a single X-Fun evaluator.

XSLT. Each template in the XSLT stylesheet is translated to a function in X-Fun. Furthermore, for each mode, we produce an additional function which implements the selection of the correct template from the set of templates associated with that mode according to their match patterns. The `call-template` and `apply-templates` instructions are translated as calls to the template or mode functions respectively. In the `copy-of` instruction, the nodes returned by the XPath expression are copied to the output using the subtree function and strings and numbers are converted to a new text node with a call to makeTree. The instructions constructing elements, attributes and other Xml nodes translate to corresponding calls to makeTree. The `for-each` instruction translates to a call to map, where the list to map over is produced by a call to evalPath and the mapping function is the body of the `for-each` instruction. Other Xslt instructions like `if` and `choose` can be translated similarly.

XQuery. The feature that most distinguishes XQuery is the Sql-like Flwor expression. It enables the programmer to create a stream of tuples using the **for** and **let** clauses, filter them with a **where** clause and then reorder them using the **order by** clause. There is no single expression in X-Fun which covers this functionality, but it is easy to build it piecewise. Using several evalPath calls we can construct the list of tuples which corresponds to the tuple stream of XQuery. Sorting and filtering of a list are functions easily definable in a functional language, and the functionality of **where** and **order by** is translated to calls to these functions. The sort and filter conditions are given again by calls to evalPath with the appropriate XPath expression. Translation of other XQuery constructs like the **if** expressions and functions proceeds in a straight forward manner.

XProc. By encapsulating each processing step in a function, X-Fun can easily express the multi-stage processing which is inherent in XProc. The pipelines then become simple function compositions. XProc steps which invoke XQuery or Xslt processing are handled by defining a function whose body is the translation of the respective program. Simple XProc steps like `split-sequence`, which splits a sequence of documents into two based on an XPath criterion are defined as normal X-Fun functions and provided as a library. The pipeline them simply calls these functions to do the required processing. The rest of the constructs like choosing among alternative subpipelines (`choose`) or looping over documents in a sequence are compiled to **match** and map expressions in X-Fun.

4.1 Compilers in more detail

On Figure 8 we show the translation of typical Xslt instructions. The X-Fun translations reference two additional functions. The first is $\text{concat}_{\mathbf{hedge}}$, which takes a list of hedges (a list of lists of trees) as an argument and concatenates them, returning a single hedge as a result. It can be defined in X-Fun in a straight forward way. The second function is toHedge, which implements the conversion from an XPath sequence to an output tree fragment. Depending on the type of items, it either calls the subtree function or converts the value to a text node. The implementation of this function is in Figure 9. For consistency with the Xslt behaviour, the function also inserts spaces between adjacent text nodes, which accounts for most of its complexity.

```
<xsl:for-each select="expression">
     ... body ...
</xsl:for-each>
                                   ⇓

concat_hedge(map_pathresult→hedge(fun cur: pathresult → hedge
     ... body ..., evalPath("expression")))
```

```
<xsl:if test="expr"> ... body ... </xsl:if>
                                   ⇓

if(evalPath("$cur[expr]") ≠ nil) then ... body ... else nil
```

```
<xsl:choose>
  <xsl:when test="expr1">... body ...</xsl:when>
  <xsl:when test="expr2">... body ...</xsl:when>
   ...
  <xsl:otherwise>... body ...</xsl:otherwise>
</xsl:choose>
                                   ⇓

if(evalPath("$cur[expr1]") ≠ nil) then ... body ...
else if(evalPath("$cur[expr2]") ≠ nil) then ... body ....
...
else ... body ...
```

```
<xsl:copy-of select="expr"/>
                                   ⇓

toHedge(evalPath("expr"))
```

```
<xsl:copy> ... body ... </xsl:copy>
                                   ⇓

makeTree(label(cur), ... body ...)
```

Fig. 8. XSLT instructions and their X-Fun equivalents

```
fun list: [pathresult]->hedge {
  match list {
    head::tail →
      (match head {
        n:node → subtree(n),
        other:bool∪number∪string →
            text(evalPath_string("string($other)"))
      })::(match list {
        (first  :bool∪number∪string)::
        (second:bool∪number∪string)::
        (rest   :[pathresult]) -> text(" ")::toHedge(tail),

        other → toHedge(tail)
      }),

    empty → nil
  }
}
```

Fig. 9. Implementation of the function toHedge.

To demonstrate the operation of our XQuery compiler, in Figure 10 we show the query Q8 from the XMark benchmark as well as the result of its automated translation to X-Fun. The translation follows the logic of the original query, which joins the Xml tables of auctions and people, and for each person displays the number of items they bought. It references two new functions. The function toString is similar to toHedge except that it converts all the items to string values. The function attribute creates an attribute node with the given name and value.

On Figure 11 we give the XProc pipeline we have used in the XProc benchmark. The X-Fun implementation of two of the mentioned XProc steps is given on Figure 12. The main X-Fun program is then simply a function composition of the given steps.

5 Implementation and Experiments

We have implemented a proof-of-concept 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, 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 an expression in all possible positions in an arbitrary order.

We have attempted to interface our implementation with Tatoo, a highly efficient evaluator of an XPath fragment based on [1]. Unfortunately, the penalty

```
for $p in /site/people/person
let $a :=
  for $t in /site/closed_auctions/closed_auction
  where $t/buyer/@person = $p/@id
  return $t
return <item person="{$p/name/text()}">{count($a)}</item>
```

$$\Downarrow$$

```
fun input: tree->hedge {
  let r = addTree(input) in
  let ca = evalPath("$r/site/closed_auctions/closed_auction") in
  map_pathresult→tree(fun p: pathresult->tree {
    element("item",
        attribute("person", toString(evalPath("$p/name/text()")))
        :: toHedge(
          let a = evalPath("$ca[buyer/@person = $p/@id]") in
          evalPath("count($a)")
        )
    )
  }, evalPath("$r/site/people/person"))
}
```

Fig. 10. XMARK query Q8 and the result of its compilation to X-Fun.

```
<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"/>
```

Fig. 11. XPROC pipeline used in the XPROC benchmark.

```
split = fun (list, test): [node] × (node → bool) → [node] × [node] {
  match list {
    head::tail →
      match split(tail, test) {
        (matched, notmatched) → if test(head)
          then (head::matched, notmatched)
          else (matched, head::notmatched)
      }
    empty → (nil, nil)
  }
}

pack = fun (first, second): [node] × [node] × label → [node] {
  match first*second {
    h::t →
      let (fh, ft) = match first {
        h::t → (toHedge(evalPath[node]("$h/node()")), t)
        empty → (nil, nil)
      } in
      let (sh, st) = match second {
        h::t → (toHedge(evalPath[node]("$h/node()")), t)
        empty → (nil, nil)
      } in
      addTree(makeTree(wrapper, fh*sh)) :: pack(ft, st, wrapper)

    empty → nil
  }
}
```

Fig. 12. Compilation of XProc steps

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, we would need to implement X-Fun in OCaml as well.

We have also implemented the compilers of Xslt and XQuery into X-Fun. 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 are fundamental and they are not implemented because of their volume. The supported fragment is wide enough to run all queries from the XMark [15] benchmark.

We don't have an XProc compiler implementation, but for the purposes of testing we have run X-Fun on manually translated programs.

5.1 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 of RAM and a SATA hard drive, running 64-bit Linux operating system.

First, we have compared the running time of our implementation on XQuery programs. We used the queries from the XMark benchmark, and the results are in Figure 13. The tests show that the running time of both tools is comparable. X-Fun is faster in case of simple queries (Q6, Q7, Q15, which contain just a simple loop), while Saxon is faster on queries involving joins (e.g., Q8, Q9, Q11). On the rest of queries our implementation of X-Fun is at most 20% slower that the competition, which we consider a good result as Saxon is a highly optimised industry tool, while we have not spent much time optimising the performance of our X-Fun implementation.

Query	X-Fun	Saxon
Q1	13.5	10.9
Q2	13.6	12.9
Q3	14.0	12.5
Q4	16.7	12.8
Q5	17.2	13.8
Q6	11.5	13.6
Q7	11.4	12.5
Q8*	962	592
Q9*	1235	705
Q10	314	222
Q11*	650	410
Q12	595	317
Q13	20.5	11.6
Q14	14.5	12.8
Q15	12.0	14.4
Q16	13.6	11.8
Q17	13.9	12.4
Q18	14.4	12.5
Q19	20.8	15.4
Q20	13.8	12.0

Fig. 13. Running time in seconds of X-Fun and Saxon on queries from the XMark benchmark on a 500 MB document. The three queries marked with '*', due to their complexity, were run on a 300 MB document.

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 4 times faster than our tool (for example, 15.7 vs. 63 seconds on a 200 MB document) and that the time of both tools scales linearly with the document size.

In the XProc comparison, we have a simple pipeline consisting of 4 steps. First, it selects subtrees from the input document, splits the resulting sequence into two based on the presence of some node. The documents from the two sequences are then joined into pairs and these pairs are concatenated to form a single document again. We have compared the performance of 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 14 and 15 (for scaling the pipeline size, we simply composed the described pipeline with itself). However, our own implementation is consistently at least two times faster, and for the larger pipelines the difference is even more apparent. 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.

Document size	X-Fun	Calabash
2 MB	8.7 s	16.6 s
4 MB	15.3 s	32.6 s
6 MB	23.1 s	51.8 s
8 MB	39.5 s	78.7 s

Fig. 14. Performance of X-Fun and Calabash on a fixed pipeline with varying input tree size

Pipeline size	X-Fun	Calabash
1	8.7 s	16.6 s
2	12 s	75.8 s
3	16 s	136.6 s
4	22 s	198.6 s

Fig. 15. Performance of X-Fun and Calabash on a 2 MB document with varying pipeline size

6 Conclusion and future work.

We have presented X-Fun, a language for processing data trees and shown that can serve as a uniform programming 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 there is a lot of space left for optimisation. We have obtained results which are a match for the Saxon's XQuery and Xslt evaluators and in the case of XProc, first results show that we are already faster than Calabash.

Our prime objective in future is to build streaming implementations of X-Fun, and thus of XQuery, Xslt, and XProc. The main ideas behind it are described in a technical report [9]. 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 [6] while streaming is not yet available for our current QuiXProc implementation.

Acknowledgement. We would like to thank Guiseppe Castagna and Kim Nguyen for their helpful discussions about the type system of X-Fun.

References

1. Arroyuelo, D., et al.: Fast in-memory xpath search using compressed indexes. In: ICDE. pp. 417–428. IEEE (2010)
2. Castagna, G., Im, H., Nguyen, K., Benzaken, V.: A Core Calculus for XQuery 3.0 (2013), `http://www.pps.univ-paris-diderot.fr/~gc/papers/xqueryduce.pdf`, unpublished manuscript
3. Frisch, A., Nakano, K.: Streaming XML transformation using term rewriting. In: Programming Language Technologies for XML (PLAN-X). pp. 2–13 (2007)
4. Fülöp, Z., Vogler, H.: Syntax-Directed Semantics – Formal Models based on Tree Transducers. EATCS Monographs in Theoretical CS, Springer (1998)
5. Hakuta, S., Maneth, S., Nakano, K., Iwasaki, H.: XQuery Streaming by Forest Transducers. In: ICDE. pp. 952–963. IEEE (2014)
6. Innovimax, INRIA Lille: Quix tools suite, `https://project.inria.fr/quix-tool-suite/`
7. Kay, M.: XSL Transformations (XSLT) Version 3.0. W3C Last Call Working Draft (2013), http://www.w3.org/TR/xslt-30
8. Kepser, S.: A simple proof for the Turing-Completeness of XSLT and XQuery. In: Extreme Markup Languages® (2004)
9. Labath, P., Niehren, J.: A Functional Language for Hyperstreaming XSLT. Research report (Mar 2013), `http://hal.inria.fr/hal-00806343`
10. Labath, P., Niehren, J.: A Uniform Programming Language for Implementing XML Standards. Research report (Jan 2015), `http://hal.inria.fr/hal-00954692`
11. Maneth, S., Berlea, A., Perst, T., Seidl, H.: XML type checking with macro tree transducers. In: PODS. pp. 283–294. ACM-Press, New York, USA (2005)
12. Neumann, A., Seidl, H.: Locating matches of tree patterns in forests. In: FSTTCS. pp. 134–145 (1998)
13. Robie, J., et al.: XQuery 3.0: An XML Query Language. W3C Proposed Recommendation (2013), http://www.w3.org/TR/xquery-30
14. Saxonica: SAXON 9.5: The XSLT and XQuery Processor, http://saxonica.com
15. Schmidt, A., et al.: XMark: A benchmark for XML data management. In: VLDB (2002), `http://www.ins.cwi.nl/projects/xmark/Assets/xmlquery.txt`
16. Walsh, N.: XML Calabash, http://xmlcalabash.com
17. Walsh, N., et al.: XProc: An XML Pipeline Language. W3C Recommendation (2010), http://www.w3.org/TR/xproc
18. Zergaoui, M., Innovimax: QuiXProc, `https://project.inria.fr/quix-tool-suite/quixproc/`