# A Declarative Embedding of XQuery in a Functional-Logic Language *

Jesús M. Almendros-Jiménez [♮], Rafael Caballero[†], Yolanda García-Ruiz[†] and Fernando Sáenz-Pérez[‡]

[♮]Dpto. Lenguajes y Computación, Universidad de Almería, Spain
[†]Dpto. de Sistemas Informáticos y Computación, UCM, Spain
[‡]Dpto. de Ingeniería del Software e Inteligencia Artificial, UCM, Spain
jalmen@ual.es, {rafa,fernan}@sip.ucm.es , ygarciar@fdi.ucm.es

**Abstract.** This paper addresses the problem of integrating a fragment of XQuery, a language for querying XML documents, into the functional-logic language $\mathcal{TOY}$. The queries are evaluated by an interpreter, and the declarative nature of the proposal allows us to prove correctness and completeness with respect to the semantics of the subset of XQuery considered. The different fragments of XML that can be produced by XQuery expressions are obtained using the non-deterministic features of functional-logic languages. As an application of this proposal we show how the typical *generate and test* techniques of logic languages can be used for generating test-cases for XQuery expressions.

## 1 Introduction

XQuery has been defined as a query language for finding and extracting information from XML [15] documents. Originally designed to meet the challenges of large-scale electronic publishing, XML also plays an important role in the exchange of a wide variety of data on the Web and elsewhere. For this reason many modern languages include libraries or encodings of XQuery, including logic programming [1] and functional programming [6]. In this paper we consider the introduction of a simple subset of XQuery [18, 20] into the functional-logic language $\mathcal{TOY}$ [11].

One of the key aspects of declarative languages is the emphasis they pose on the logic semantics underpinning declarative computations. This is important for reasoning about computations, proving properties of the programs or applying declarative techniques such as abstract interpretation, partial evaluation or algorithmic debugging [14]. There are two different declarative alternatives that can be chosen for incorporating XML into a (declarative) language:

1. Use a domain-specific language and take advantage of the specific features of the host language. This is the approach taken in [9], where a rule-based

* Work partially supported by the Spanish projects STAMP TIN2008-06622-C03-01, DECLARAWEB TIN2008-06622-C03-03, Prometidos-CM S2009TIC-1465 and GPD UCM-BSCH-GR58/08-910502.

language for processing semi-structured data that is implemented and embedded into the functional-logic language Curry, and also in [13] for the case of logic programming.
2. Consider an existing query language such as XQuery, and embed a fragment of the language in the host language, in this case $\mathcal{TOY}$. This is the approach considered in this paper.

Thus, our goal is to include XQuery using the purely declarative features of the host languages. This allows us to prove that the semantics of the fragment of XQuery has been correctly included in $\mathcal{TOY}$. To the best of our knowledge, it is the first time a fragment of XQuery has been encoded in a functional-logic language. A first step in this direction was proposed in [5], where XPath [16] expressions were introduced in $\mathcal{TOY}$. XPath is a subset of XQuery that allows navigating and returning fragments of documents in a similar way as the path expressions used in the *chdir* command of many operating systems. The contributions of this paper with respect to [5] are:
- The setting has been extended to deal with a simple fragment of XQuery, including *for* statements for traversing XML sequences, *if/where* conditions, and the possibility of returning XML elements as results. Some basic XQuery constructions such as *let* statements are not considered, but we think that the proposal is powerful enough for representing many interesting queries.
- The soundness of the approach is formally proved, checking that the semantics of the fragment of XQuery is correctly represented in $\mathcal{TOY}$.

Next section introduces the fragment of XQuery considered and a suitable operational semantics for evaluating queries. The language $\mathcal{TOY}$ and its semantics are presented in Section 3. Section 4 includes the interpreter that performs the evaluation of simple XQuery expressions in $\mathcal{TOY}$. The theoretical results establishing the soundness of the approach with respect to the operational semantics of Section 2 are presented in Section 4.1. Section 5 explains the automatic generation of test cases for simple XQuery expressions. Finally, Section 6 concludes summarizing the results and proposing future work.

An extended version of the paper including proofs of the theoretical results can be found at [2].

## 2 XQuery and its Operational Semantics

XQuery allows the user to query several documents, applying join conditions, generating new XML fragments, and using many other features [18, 20]. The syntax and semantics of the language are quite complex [19], and thus only a small subset of the language is usually considered. The next subsection introduces the fragment of XQuery considered in this paper.

### 2.1 The subset SXQ

In [4] a declarative subset of XQuery, called XQ, is presented. This subset is a core language for XQuery expressions consisting of *for*, *let* and *where/if* statements.



$$
\begin{aligned}
query &::= (\,) \mid query\ query \mid tag\\
&\mid\ doc(File) \mid doc(File)/axis :: \nu \mid var \mid var/axis :: \nu\\
&\mid\ \mathtt{for}\ var\ \mathtt{in}\ query\ \mathtt{return}\ query\\
&\mid\ \mathtt{if}\ cond\ \mathtt{then}\ query\\
cond &::= var{=}var \mid query\\
tag &::= \langle a\rangle\ var \ldots var\ \langle /a\rangle \mid \langle a\rangle\ tag\ \langle /a\rangle
\end{aligned}
$$

**Fig. 1.** Syntax of SXQ, a simplified version of XQ

In this paper we consider a simplified version of XQ which we call SXQ and whose syntax can be found in Figure 1, where *axis* can be one of *child*, *self*, *descendant* or *dos* (i.e. descendant or self), and $\nu$ is a node test. The differences of SXQ with respect to XQ are:

1. XQ includes the possibility of using variables as tag names using a constructor *lab($x)*.
2. XQ permits enclosing any query $Q$ between tag labels $\langle a\rangle Q\langle /a\rangle$. SXQ only admits either variables or other tags inside a tag.

Our setting can be easily extended to support the *lab($x)* feature, but we omit this case for the sake of simplicity in this presentation. The second restriction is more severe: although *lets* are not part of XQ, they could be simulated using *for* statements inside tags. In our case, forbidding other queries different from variables inside tag structures imply that our core language cannot represent *let* expressions. This limitation is due to the non-deterministic essence of our embedding, since a *let* expression means collecting all the results of a query instead of producing them separately using non-determinism. In spite of these limitations, the language SXQ is still useful for solving many common queries as the following example shows.

*Example 1.* Consider an XML file *"bib.xml"* containing data about books, and another file *"reviews.xml"* containing reviews for some of these books (see [17], sample data 1.1.2 and 1.1.4 to check the structure of these documents and an example). Then we can list the reviews corresponding to books in *"bib.xml"* as follows:

```
for $b in doc("bib.xml")/bib/book,
    $r in doc("reviews.xml")/reviews/entry
where $b/title = $r/title
for $booktitle in $r/title,
    $revtext in $r/review
return <rev> $booktitle  $revtext </rev>
```

The variable $b takes the value of the different books, and $r the different reviews. The *where* condition ensures that only reviews corresponding to the book are considered. Finally, the last two variables are only employed to obtain the book title and the text of the review, the two values that are returned as output of the query by the *return* statement.

It can be argued that the code of this example does not follow the syntax of Figure 1. While this is true, it is very easy to define an algorithm that converts a query formed by *for*, *where* and *return* statements into a SXQ query (as long as it only includes variables inside tags, as stated above). The idea is simply to convert the *where* into *ifs*, following each *for* by a *return*, and decomposing XPath expressions including several steps into several *for* expressions by introducing a new auxiliary variable and each one consisting of a single step.

*Example 2.* The query of Example 1 using SXQ syntax:

```
for $x1 in doc("bib.xml")/child::bib return
for $x2 in $x1/child::book  return
for $x3 in doc("reviews.xml")/child::reviews return
for $x4 in $x3/entry  return
if ($x2/title = $x4/title) then
    for $x5 in $x4/title return
     for $x6 in $x4/review return  <rev> $x5  $x6 </rev>
```

We end this subsection with a few definitions that are useful for the rest of the paper. The set of variables in a query $Q$ is represented as $Var(Q)$. Given a query $Q$, we use the notation $Q_{|p}$ for representing the subquery $Q'$ that can be found in $Q$ at position $p$. Positions are defined as usual in syntax trees:

**Definition 1.** *Given a query $Q$ and a position $p$, $Q_{|p}$ is defined as follows:*

$$\begin{array}{ll}
Q_{|\varepsilon} & = Q \\
(Q_1\ Q_2)_{|(i\cdot p)} & = (Q_i)_{|p}\ \ i \in \{1,2\} \\
(\texttt{for var in } Q_1 \texttt{ return } Q_2)_{|(i\cdot p)} & = (Q_i)_{|p}\ \ i \in \{1,2\} \\
(\texttt{if } Q_1 \texttt{ then } Q_2)_{|(i\cdot p)} & = (Q_i)_{|p}\ \ i \in \{1,2\} \\
(\texttt{if var} = \texttt{var then } Q_1)_{|(1\cdot p)} & = (Q_1)_{|p}
\end{array}$$

Hence the position of a subquery is the path in the syntax tree represented as the concatenation of children positions $p_1 \cdot p_2 \ldots \cdot p_n$. For every position $p$, $\varepsilon \cdot p = p \cdot \varepsilon = p$. In general $Q_{|p}$ is not a proper SXQ query, since it can contain *free variables*, which are variables defined previously in *for* statements in $Q$. The set of variables of $Q$ that are *relevant* for $Q_{|p}$ is the subset of $Var(Q)$ that can appear free in any subquery at position $p$. This set, denoted as $Rel(Q,p)$ is defined recursively as follows:

**Definition 2.** *Given a query $Q$, and a position $p$, $Rel(Q,p)$ is defined as:*

1. $\emptyset$, *if* $p = \varepsilon$ .
2. $Rel(Q_1, p')$, *if* $Q \equiv Q_1\ Q_2$, $p = 1 \cdot p'$.
3. $Rel(Q_2, p')$, *if* $Q \equiv Q_1\ Q_2$, $p = 2 \cdot p'$.
4. $Rel(Q_1, p')$, *if* $Q \equiv$ `for var in` $Q_1$ `return` $Q_2$, $p = 1 \cdot p'$.
5. $\{\text{var}\} \cup Rel(Q_2, p')$, *if* $Q \equiv$ `for var in` $Q_1$ `return` $Q_2$, $p = 2 \cdot p'$.
6. $Rel(Q_1, p')$, *if* $Q \equiv$ `if` $Q_1$ `then` $Q_2$, $p = 1 \cdot p'$.
7. $Rel(Q_2, p')$, *if* $Q \equiv$ `if` $Q_1$ `then` $Q_2$, $p = 2 \cdot p'$.

*Observe that cases $Q \equiv ()$, $Q \equiv tag$, $Q \equiv \text{var}$, $Q \equiv \text{var}/\chi :: \nu$, and $\text{var} = \text{var}$ correspond to $p \equiv \varepsilon$.*

Without loss of generality we assume that all the relevant variables for a given position are indexed starting from 1 at the outer level. We also assume that every `for` statement introduces a new variable. A query like `for X in ((for Y in ...) (for Y in ...)) ...` is then renamed to an equivalent query of the form `for` $X_1$ `in ((for` $X_2$ `in ...) (for` $X_3$ `in ...)) ...` (notice that the two $Y$ variables occurred in different scopes).

## 2.2 XQ Operational Semantics

Figure 2 introduces the operational semantics of XQ that can be found in [4]. The only difference with respect to the semantics of this paper is that there is no rule for the constructor *lab*, for the sake of simplicity.

$$\begin{array}{rl}
[\![()]\!]_k(\mathcal{F},\bar{e}) & = (\mathcal{F},[\,]) \\
[\![Q_1\ Q_2]\!]_k(\mathcal{F},\bar{e}) & = [\![Q_1]\!]_k(\mathcal{F},\bar{e}) \uplus [\![Q_2]\!]_k(\mathcal{F},\bar{e}) \\
[\![\texttt{for } \$x_{k+1} \texttt{ in } Q_1 \texttt{ return } Q_2]\!]_k(\mathcal{F},\bar{e}) & = \text{let } (\mathcal{F}',\bar{l}) = [\![Q_1]\!]_k(\mathcal{F},\bar{e}) \text{ in} \\
& \quad \biguplus_{1\le i\le |\bar{l}|} [\![Q_2]\!]_{k+1}(\mathcal{F}',\bar{e}\cdot l_i) \\
[\![\$x_i]\!]_k(\mathcal{F},[e_1,\ldots,e_k]) & = (\mathcal{F},[e_i]) \\
[\![\$x_i/\chi :: \nu]\!]_k(\mathcal{F},[e_1,\ldots,e_k]) & = (\mathcal{F}, \text{list of nodes } v \text{ such that } \chi^{\mathcal{F}}(e_i,v) \text{ and} \\
& \quad \text{label name of } v = \nu \text{ in order } <_{doc}^{tree(e_i)}) \\
[\![\texttt{if } C \texttt{ then } Q_1]\!]_k(\mathcal{F},\bar{e}) & = \text{if } \pi_2([\![C]\!]_k(\mathcal{F},\bar{e})) \neq [\,] \text{ then } [\![Q_1]\!]_k(\mathcal{F},\bar{e}) \\
& \quad \text{else } (\mathcal{F},[\,]) \\
[\![\$x_i = \$x_j]\!]_k(\mathcal{F},[e_1,\ldots,e_k]) & = \text{if } e_i = e_j \text{ then } construct(yes,(\mathcal{F},[\,])) \\
& \quad \text{else } (\mathcal{F},[\,])
\end{array}$$

**Fig. 2.** Semantics of Core XQuery

As explained in [4], the previous semantics defines the denotation of an XQ expression $Q$ with $k$ relevant variables, under a graph-like representation of a data forest $\mathcal{F}$, and a list of indexes $\bar{e}$ in $\mathcal{F}$, denoted by $[\![Q]\!]_k(\mathcal{F},\bar{e})$. In particular, each relevant variable $\$x_i$ of $Q$ has as value the tree of $\mathcal{F}$ indexed at position $e_i$. $\chi^{\mathcal{F}}(e_i,v)$ is a boolean function that returns *true* whenever $v$ is the subtree of $\mathcal{F}$ indexed at position $e_i$. The operator $construct(a,(\mathcal{F},[w_1...w_n]))$, denotes the construction of a new tree, where $a$ is a label, $\mathcal{F}$ is a data forest, and $[w_1\ldots w_n]$ is a list of nodes in $\mathcal{F}$. When applied, *construct* returns an indexed forest $(\mathcal{F} \cup T', [root(T')])$, where $T'$ is a tree with domain a new set of nodes, whose root is labeled with $a$, and with the subtree rooted at the $i$-th (in sibling order) child of root($T'$) being an isomorphic copy of the subtree rooted by $w_i$ in $\mathcal{F}$. The symbol $\uplus$ used in the rules takes two indexed forests $(\mathcal{F}_1,l_1)$, $(\mathcal{F}_2,l_2)$ and returns an indexed forest $(\mathcal{F}_1 \cup \mathcal{F}_2, l)$, where $l = l_1 \cdot l_2$. Finally, $tree(e_i)$ denotes the maximal tree within the input forest that contains the node $e_i$, hence $<_{doc}^{tree(e_i)}$ is the document order on the tree containing $e_i$.

Without loss of generality this semantics assumes that all the variables relevant for a subquery are numbered consecutively starting by 1 as in Example 2. It also assumes that the documents appear explicitly in the query. That is, in Example 2 we must suppose that instead of *doc("bib.xml")* we have the XML corresponding to this document. Of course this is not feasible in practice, but simplifies the theoretical setting and it is assumed in the rest of the paper.

These semantic rules constitute a term rewriting system (TRS in short, see [3]), with each rule defining a single reduction step. The symbol $:=^*$ represents the reflexive and transitive closure of $:=$ as usual. The TRS is terminating and confluent (the rules are not overlapping). Normal forms have the shape $(\mathcal{F}, e_1, \ldots, e_n)$ where $\mathcal{F}$ is a forest of XML fragments, and $e_i$ are nodes in $\mathcal{F}$, meaning that the query returns the XML fragments (**indexed by**) $e_1, \ldots, e_n$. The semantics evaluates a query starting with the expression $[\![Q]\!]_0(\emptyset, ())$. Along intermediate steps, expressions of the form $[\![Q']\!]_k(\mathcal{F}, \bar{e}_k)$ are obtained. The idea is that $Q'$ is a subquery of $Q$ with $k$ relevant variables (which can occur free in $Q'$), that must take the values $\bar{e}_k$. The next result formalizes these ideas.

**Proposition 1.** *Let $Q$ be a SXQ query. Suppose that*

$$[\![Q]\!]_0(\emptyset, ()) :=^* [\![Q']\!]_n(\mathcal{F}, \bar{e}_n)$$

*Then:*

- *$Q'$ is a subquery of $Q$, that is, $Q' = Q_p$ for some $p$.*
- *$Rel(Q, p) = \{X_1, \ldots, X_n\}$.*
- *Let $S$ be the set of free variables in $Q'$. Then $S \subset Rel(Q, p)$.*
- *$[\![Q']\!]_n(\mathcal{F}, \bar{e}_n) = [\![Q'\theta]\!]_0(\emptyset, ())$, with $\theta = \{X_1 \mapsto e_1, \ldots, X_n \mapsto e_n\}$*

*Proof.* Straightforward from Definition 2, and from the XQ semantic rules of Figure 2.

A more detailed discussion about this semantics and its properties can be found in [4].

## 3 $\mathcal{TOY}$ and Its Semantics

A $\mathcal{TOY}$ [11] program is composed of data type declarations, type alias, infix operators, function type declarations and defining rules for functions symbols. The syntax of *partial expressions* in $\mathcal{TOY}$ $e \in Exp_\perp$ is $e ::= \perp \mid X \mid h \mid (e\ e')$ where $X$ is a variable and $h$ either a function symbol or a data constructor. Expressions of the form $(e\ e')$ stand for the application of expression $e$ (acting as a function) to expression $e'$ (acting as an argument). Similarly, the syntax of *partial patterns* $t \in Pat_\perp \subset Exp_\perp$ can be defined as $t ::= \perp \mid X \mid c\ t_1 \ldots t_m \mid f\ t_1 \ldots t_m$ where $X$ represents a variable, $c$ a data constructor of arity greater or equal to



$m$, and $f$ a function symbol of arity greater than $m$, being $t_i$ partial patterns for all $1 \leq i \leq m$. Each rule for a function $f$ in $\mathcal{TOY}$ has the form:

$$\underbrace{f\ t_1 \ldots t_n}_{\text{left-hand side}} \rightarrow \underbrace{r}_{\text{right-hand side}} \Leftarrow \underbrace{C_1, \ldots, C_k}_{\text{condition}} \text{ where } \underbrace{s_1 = u_1, \ldots, s_m = u_m}_{\text{local definitions}}$$

where $u_i$ and $r$ are expressions (that can contain new extra variables), $C_j$ are strict equalities, and $t_i$, $s_i$ are patterns. In $\mathcal{TOY}$, variable names must start with either an uppercase letter or an underscore (for anonymous variables), whereas other identifiers start with lowercase.

Data type declarations and type alias are useful for representing XML documents in $\mathcal{TOY}$:

```
data node      = txt     string
               | comment string
               | tag     string [attribute] [node]
data attribute = att     string string
type xml       = node
```

The data type `node` represents nodes in a simple XML document. It distinguishes three types of nodes: texts, tags (element nodes), and comments, each one represented by a suitable data constructor and with arguments representing the information about the node. For instance, constructor `tag` includes the tag name (an argument of type `string`) followed by a list of attributes, and finally a list of child nodes. The data type `attribute` contains the name of the attribute and its value (both of type `string`). The last type alias, `xml`, renames the data type `node`. Of course, this list is not exhaustive, since it misses several types of XML nodes, but it is enough for this presentation.

$\mathcal{TOY}$ includes two primitives for loading and saving XML documents, called `load_xml_file` and `write_xml_file` respectively. For convenience all the documents are started with a dummy node `root`. This is useful for grouping several XML fragments. If the file contains only one node `N` at the outer level, the `root` node is unnecessary, and can be removed using this simple function:

```
load_doc F = N <== load_xml_file F == xmlTag "root" [] [N]
```

where `F` is the name of the file containing the document. Observe that the strict equality `==` in the condition forces the evaluation of `load_xml_file F` and succeeds if the result has the form `xmlTag "root" [] [N]` for some `N`. If this is the case, `N` is returned.

The constructor-based ReWriting Logic (CRWL) [7] has been proposed as a suitable declarative semantics for functional-logic programming with lazy non-deterministic functions. The calculus is defined by five inference rules (see Figure 3): (BT) that indicates that any expression can be approximated by bottom, (RR) that establishes the reflexivity over variables, the decomposition rule (DC), the (JN) (join) rule that indicates how to prove strict equalities, and the function application rule (FA). In every inference rule, $r, e_i, a_j \in Exp_\perp$ are partial expres-

$$\begin{array}{lll}
\text{BT} & e \to \perp & \\
\text{RR} & X \to X & \text{with } X \in Var \\
\text{DC} & \dfrac{e_1 \to t_1 \ \dots\ e_m \to t_m}{h\,\overline{e}_m \to h\,\overline{t}_m} & h\,\overline{t}_m \in Pat_\perp \\
\text{JN} & \dfrac{e \to t \quad e' \to t}{e == e'} & t \in Pat \text{ (total pattern)} \\
\text{FA} & \dfrac{e_1 \to t_1 \ \dots\ e_n \to t_n \quad C \quad r\,\overline{a}_k \to t}{f\,\overline{e}_n\,\overline{a}_k \to t} & \\
& \text{if } (f\,\overline{t}_n \to r \Leftarrow C) \in [P]_\perp, t \neq \perp &
\end{array}$$

**Fig. 3.** CRWL Semantic Calculus

sions and $t, t_k \in Pat_\perp$ are partial patterns. The notation $[P]_\perp$ of the inference rule $FA$ represents the set $\{(l \to r \Leftarrow C)\theta \mid (l \to r \Leftarrow C) \in P,\ \theta \in Subst_\perp\}$ of partial instances of the rules in program $P$ ($Subst_\perp$ represents the set of partial substitutions that replace variables by partial terms). The most complex inference rule is $FA$ (Function Application), which formalizes the steps for computing a *partial pattern* $t$ as approximation of a function call $f\,\overline{e}_n$:

1. Obtain partial patterns $t_i$ as suitable approximations of the arguments $e_i$.
2. Apply a program rule $(f\,\overline{t}_n \to r \Leftarrow C) \in [P]_\perp$, verify the condition $C$, and check that $t$ approximates the right-hand side $r$.

In this semantic notation, local declarations $a = b$ introduced in $\mathcal{TOY}$ syntax by the reserved word `where` are part of the condition $C$ as approximation statements of the form $b \to a$.
The semantics in $\mathcal{TOY}$ allows introducing non-deterministic functions, such as the following function `member` that returns all the elements in a list:

```
member:: [A]  -> A
member  [X | Xs] =  X
member  [X | Xs] =  member Xs
```

Another example of $\mathcal{TOY}$ function is the definition of the infix operator `.::.` for XPath expressions (the operator `::` in XPath syntax):

```
(.::.) :: (A -> B) -> (B -> C) -> (A -> C)
(F .::. G) X = G (F X)
```

As the examples show, $\mathcal{TOY}$ is a typed language. However the type declaration is optional and in the rest of the paper they are omitted for the sake of simplicity. *Goals* in $\mathcal{TOY}$ are sequences of strict equalities. A strict equality $e_1 == e_2$ holds (inference $JN$) if both $e_1$ and $e_2$ can be reduced to the same total pattern $t$. For instance, the goal `member [1,2,3,4] == R` yields four answers, the four values for `R` that make the equality true: {`R` $\mapsto$ `1`}, ..., {`R` $\mapsto$ `4`}.



## 4 Transforming SXQ into $\mathcal{TOY}$

In order to represent SXQ queries in $\mathcal{TOY}$ we use some auxiliary datatypes:

```
type xPath = xml-> xml

data sxq  = xfor xml sxq sxq | xif cond sxq | xmlExp xml |
                xp path | comp sxq sxq
data cond = xml := xml | cond sxq
data path = var xml | xml :/ xPath | doc string xPath
```

The structure of the datatype `sxq` allows representing any SXQ query (see SXQ syntax in Figure 1). It is worth noticing that a variable introduced by a *for* statement has type `xml`, indicating that the variable always contains a value of this type. $\mathcal{TOY}$ includes a primitive `parse_xquery` that translates any SXQ expression into its corresponding representation as a term of this datatype, as the next example shows:

*Example 3.* The translation of the SXQ query of Example 2 into the datatype `sxq` produces the following $\mathcal{TOY}$ data term:

```
Toy> parse_xquery  "for $x1 in doc(\"bib.xml\")/child::bib return
          for $x2 in .....  <rev> $x5  $x6 </rev>" == R
yes
{R --> xfor X1 (xp (doc "bib.xml" (child .::. (nameT "bib"))))
    (xfor X2 (xp ( X1 :/ (child .::.(nameT "book"))))
    (xfor X3 (xp (doc "reviews.xml" (child  .::. (nameT "reviews"))))
    (xfor X4 (xp ( X3 :/ (child .::.(nameT "entry"))))
    (xif  ((xp(X2 :/ (child .::.(nameT "title")))) :=
             (xp(X4 :/ (child .::.(nameT "title")))))
    (xfor X5 (xp ( X4 :/ (child .::.(nameT "title"))))
    (xfor X6 (xp ( X4 :/ (child .::.(nameT "review"))))
       (xmlExp (xmlTag "rev" [] [X5,X6]))))))))
}
```

The interpreter assumes the existence of the infix operator `.::.` that connects axes and tests to build steps, defined as the sequence of applications in Section 3.

The rules of the $\mathcal{TOY}$ interpreter that processes SXQ queries can be found in Figure 4. The main function is `sxq`, which distinguishes cases depending of the form of the query. If it is an XPath expression then the auxiliary function `sxqPath` is used. If the query is an XML expression, the expression is just returned (this is safe thanks to our constraint of allowing only variables inside XML expressions). If we have two queries (`comp` construct), the result of evaluating any of them is returned using non-determinism. The `for` statement (`xfor` construct) forces the evaluation of the query `Q1` and binds the variable `X` to the result. Then the result query `Q2` is evaluated. The case of the `if` statement is analogous. The XPath subset considered includes tests for attributes (`attr`), label names (`nameT`), general elements (`nodeT`) and text nodes (`textT`). It also

```
sxq (xp E)             = sxqPath E
sxq (xmlExp X)         = X
sxq (comp Q1 Q2)       = sxq Q1
sxq (comp Q1 Q2)       = sxq Q2
sxq (xfor X Q1 Q2)     = sxq Q2 <== X== sxq Q1
sxq (xif (Q1:=Q2) Q3)  = sxq Q3 <== sxq Q1  == sxq Q2
sxq (xif (cond Q1) Q2) = sxq Q2 <== sxq Q1  == _

sxqPath (var X)   = X
sxqPath (X :/ S) = S X
sxqPath (doc F S) = S (load_xml_file F)

%%%% XPATH %%%%%%
attr A (xmlTag S Attr L ) = xmlText T <== member Attr == xmlAtt A T
nameT S (xmlTag S Attr L ) = xmlTag S Attr L
nodeT X = X
textT (xmlText S) = xmlText S
commentT S (xmlComment S) = xmlComment S

self X = X
child (xmlTag _Name _Attr L) = member L
descendant X = child X
descendant X = descendant Y <== child X == Y
dos = self
dos = descendant
```

**Fig. 4.** $\mathcal{TOY}$ transformation rules for SXQ

includes the axes **self**, **child**, **descendant** and **dos**. Observe that we do not include reverse axes like **ancestor** because they can be replaced by expressions including forward axes, as shown in [12]. Other constructions such as filters can be easily included (see [5]). The next example uses the interpreter to obtain the answers for the query of our running example.

*Example 4.* The goal `sxq (parse_xquery "for....")) == R` applies the interpreter of Figure 4 to the code of Example 2 (assuming that the string after `parse_xquery` is the query in Example 2), and returns the $\mathcal{TOY}$ representation of the expected results:

```
<rev>
 <title>TCP/IP Illustrated</title>
 <review>  One of the best books on TCP/IP.  </review>
</rev>
...
```

### 4.1 Soundness of the Transformation

One of the goals of this paper is to ensure that the embedding is semantically correct and complete. This section introduces the theoretical results establishing



these properties. If $V$ is a set of indexed variables of the form $\{X_1, \ldots, X_n\}$ we use the notation $\theta(V)$ to indicate the sequence $\theta(X_1), \ldots, \theta(X_n)$. In these results it is implicitly assumed that there is a bijective mapping $f$ from XML format to the datatype `xml` in $\mathcal{TOY}$. Also, variables in XQuery $\$x_i$ are assumed to be represented in $\mathcal{TOY}$ as $X_i$ and conversely. However, in order to simplify the presentation, we omit the explicit mention to $f$ and to $f^{-1}$.

**Lemma 1.** *Let $P$ be a $\mathcal{TOY}$ program, $Q'$ an SXQ query, and $Q, p$ such that $Q \equiv Q'_{|p}$. Define $V = Rel(Q', p)$ (see Definition 2), and $k = |V|$. Let $\theta$ be a substitution such that $\mathcal{P} \vdash$ (`sxq` $Q\theta == t$) for some pattern $t$.*
*Then $[\![Q]\!]_k(\mathcal{F}, [\theta(V)]) :=^* (\mathcal{F}', L)$, for some forests $\mathcal{F}, \mathcal{F}'$ and with $L$ verifying $t \in L$.*

The theorem that establishes the correctness of the approach is an easy consequence of the Lemma.

**Theorem 1.** *Let $P$ be the $\mathcal{TOY}$ program of Figure 4, $Q$ an SXQ query, $t$ a $\mathcal{TOY}$ pattern, and $\theta$ a substitution such that $\mathcal{P} \vdash$ (`sxq` $Q\theta == t$) for some $\theta$. Then $[\![Q]\!]_0(\emptyset, []) :=^* (\mathcal{F}, L)$, for some forest $\mathcal{F}$, and $L$ verifying $t \in L$.*

*Proof.* In Lemma 1 consider the position $p \equiv \varepsilon$. Then $Q' \equiv Q$, $V = \emptyset$ and $k = 0$. Without loss of generality we can restrict in the conclusion to $\mathcal{F} = \emptyset$, because $\theta(V) = \emptyset$ and therefore $\mathcal{F}$ is not used during the rewriting process. Then the conclusion of the theorem is the conclusion of the lemma.

Thus, our approach is correct. The next Lemma allows us to prove that it is also complete, in the sense that the $\mathcal{TOY}$ program can produce every answer obtained by the XQ operational semantics.

**Lemma 2.** *Let $\mathcal{P}$ be the $\mathcal{TOY}$ program of Figure 4. Let $Q'$ be a SXQ query and $Q$, $p$ such that $Q \equiv Q'_{|p}$. Define $V = Rel(Q', p)$ (see Definition 2) and $k = |V|$. Suppose that $[\![Q]\!]_k(\mathcal{F}, \overline{e}_k) :=^* (\mathcal{F}', \overline{a}_n)$ for some $\mathcal{F}$, $\mathcal{F}'$, $\overline{e}_k$, $\overline{a}_n$.*
*Then, for every $a_j$, $1 \le j \le n$, there is a substitution $\theta$ such that $\theta(X_i) = e_i$ for $X_i \in V$ and a CRWL-proof proving $\mathcal{P} \vdash$ `sxq` $Q\theta == a_j$.*

As in the case of correctness, the completeness theorem is just a particular case of the Lemma:

**Theorem 2.** *Let $\mathcal{P}$ be the $\mathcal{TOY}$ program of Figure 4. Let $Q$ be a SXQ query and suppose that $[\![Q]\!]_k(\emptyset, []) :=^* (\mathcal{F}, \overline{a}_n)$ for some $\mathcal{F}$, $\overline{a}_n$. Then for every $a_j$, $1 \le j \le n$, there is $\mathcal{P} \vdash$ (`sxq` $Q)\theta == a_j$ for some substitution $\theta$.*

*Proof.* As in Theorem 1, suppose $p \equiv \varepsilon$ and thus $Q' \equiv Q$. Then $V = \emptyset$ and $k = 0$. Then, if $[\![Q]\!]_0(\emptyset, \emptyset) :=^* (\mathcal{F}, \overline{a}_n)$ it is easy to check that $[\![Q]\!]_0(\mathcal{F}', \emptyset) :=^* (\mathcal{F}, \overline{a}_n)$ for any $\mathcal{F}'$. Then the conclusion of the lemma is the same as the conclusion of the Theorem.

The proofs of Lemmata 1 and 2 can be found in [2].

## 5 Application: Test Case Generation

In this section we show how an embedding of SXQ into $\mathcal{TOY}$ can be used for obtaining test-cases for the queries. For instance, consider the erroneous query of the next example.

*Example 5.* Suppose that the user also wants to include the publisher of the book among the data obtained in Example 1. The following query tries to obtain this information:

```
Q = for $b in doc("bib.xml")/bib/book,
        $r in doc("reviews.xml")/reviews/entry,
        where $b/title = $r/title
        for $booktitle in $r/title,
            $revtext in $r/review,
            $publisher in $r/publisher
        return  <rev> $booktitle $publisher $revtext </rev>
```

However, there is an error in this query, because in `$r/publisher` the variable `$r` should be `$b`, since the publisher is in the document *"bib.xml"*, not in *"reviews.xml"*. The user does not notice that there is an error, tries the query (in $\mathcal{TOY}$ or in any XQuery interpreter) and receives an empty answer.

In order to check whether a query is erroneous, or even to help finding the error, it is sometimes useful to have test-cases, i.e., XML files which can produce some answer for the query. Then the test-cases and the original XML documents can be compared, and this can help finding the error. In our setting, such test-cases are obtained for free, thanks to the generate and test capabilities of logic programming. The general process can be described as follows:

1. Let `Q'` be the translation `parse_xquery Q` of query `Q` into $\mathcal{TOY}$.
2. Let $F_1, \ldots, F_k$ be the names of the XML documents occurring in `Q'`. That is, for each $F_i$, $1 \leq i \leq k$, there is an occurrence of an expression of the form `load_xml_file(`$F_i$`)` in `Q'` (which corresponds to expressions `doc(`$F_i$`)` in `Q`). Let `Q''` be the result of replacing each `doc(`$F_i$`)` expression by a new variable $D_i$, for $i = 1 \ldots k$.
3. Let "`expected.xml`" be a document containing an expected answer for the query `Q`.
4. Let `E` the expression `Q''==load_doc "expected.xml"`.
5. Try the goal
   `G` $\equiv$ `E, write_xml_file` $D_1$ $F'_1$`, ..., write_xml_file` $D_k$ $F'_k$

The idea is that the goal `G` looks for values of the logic variables $D_i$ fulfilling the strict equality. The result is that after solving this goal, the $D_i$ variables contain XML documents that can produce the expected answer for this query. Then each document is saved into a new file with name $F'_i$. For instance $F'_i$ can consist of the original name $F_i$ preceded by some suitable prefix *tc*. The process can be automatized, and the result is the code of Figure 5.



```
prepareTC (xp E)            = (xp E',L)
                              where (E',L) = prepareTCPath E
prepareTC (xmlExp X)        = (xmlExp X, [])
prepareTC (comp Q1 Q2)      = (comp Q1' Q2', L1++L2)
                              where (Q1',L1) = prepareTC Q1
                                    (Q2',L2) = prepareTC Q2
prepareTC (xfor X Q1 Q2)    = (xfor X Q1' Q2', L1++L2)
                              where (Q1',L1) = prepareTC Q1
                                    (Q2',L2) = prepareTC Q2
prepareTC (xif (Q1:=Q2) Q3) = (xif (Q1':=Q2') Q3',L1++(L2++L3))
                              where (Q1',L1) = prepareTC Q1
                                    (Q2',L2) = prepareTC Q2
                                    (Q3',L3) = prepareTC Q3
prepareTC (xif (cond Q1) Q2) = (xif (cond Q1) Q2, L1++L2)
                              where (Q1',L1) = prepareTC Q1
                                    (Q2',L2) = prepareTC Q2

prepareTCPath (var X)    = (var X, [])
prepareTCPath (X :/ S)   = (X :/ S, [])
prepareTCPath (doc F S)  = (A :/ S, [write_xml_file A ("tc"++F)])

generateTC Q F = true <== sxq Qtc == load_doc F, L==_
           where (Qtc,L) = prepareTC Q
```

**Fig. 5.** $\mathcal{TOY}$ transformation rules for SXQ

The code uses the list concatenation operator ++ which is defined in $\mathcal{TOY}$ as usual in functional languages such as Haskell. It is worth observing that if there are no test-case documents that can produce the expected result for the query, the call to `generateTC` will loop. The next example shows the generation of test-cases for the wrong query of Example 5.

*Example 6.* Consider the query of Example 5 , and suppose the user writes the following document "`expected.xml`":

```
<rev>
 <title>Some title</title>
 <review>The review</review>
 <publisher>Publisher</publisher>
</rev>
```

This is a possible expected answer for the query. Now we can try the goal:

```
Toy> Q == parse_xquery "for....", R == generateTC Q "expected.xml"
```

The first strict equality parses the query, and the second one generates the XML documents which constitute the test cases. In this example the test-cases obtained are:

```
% bibtc.xml
<bib>
 <book>
  <title>Some title</title>
 </book>
</bib>
```

```
% revtc.xml
<reviews>
 <entry>
  <title>Some title</title>
  <review>The review </review>
  <publisher>Publisher</publisher>
 </entry>
</reviews>
```

By comparing the test-case "revtc.xml" with the file "reviews.xml" we observe that the publisher is not in the reviews. Then it is easy to check that in the query the publisher is obtained from the reviews instead of from the *bib* document, and that this constitutes the error.

## 6 Conclusions

The paper shows the embedding of a fragment of the XQuery language for querying XML documents into the functional-logic language $\mathcal{TOY}$. Although only a small subset of XQuery consisting of *for*, *where/if* and *return* statements has been considered, the users of $\mathcal{TOY}$ can now perform simple queries such as *join* operations. The formal definition of the embedding allows us to prove the soundness of the approach with respect to the operational semantics of XQuery. The proposal respects the declarative nature of $\mathcal{TOY}$, exploiting its non-deterministic nature for obtaining the different results produced by XQuery expressions. An advantage of this approach with respect to the use of lists usually employed in functional languages is that our embedding allows the user to generate test-cases automatically when possible, which is useful for testing the query, or even for helping to find the error in the query. An extended version of this paper, including the proofs of the theoretical results and more detailed explanations about how to install $\mathcal{TOY}$ and run the prototype can be found in [2].

The most obvious future work would be introducing the *let* statement, which presents two novelties. The first is that they are *lazy*, that is, they are not evaluated if they are not required by the result. This part is easy to fulfill since we are in a lazy language. In particular, they could be introduced as local definitions (*where* statements in $\mathcal{TOY}$). The second novelty is more difficult to capture, and it is that the variables introduced by *let* represent an XML sequence. The natural representation in $\mathcal{TOY}$ would be a list, but the non-deterministic nature of our proposal does not allow us to collect all the results provided by an expression in a declarative way. A possible idea would be to use the functional-logic language Curry [8] and its encapsulated-search [10], or even the non-declarative *collect* primitive included in $\mathcal{TOY}$. In any case, this will imply a different theoretical framework and new proofs for the results. A different line for future work is the use of test cases for finding the error in the query using some variation of declarative debugging [14] that could be applied to this setting.



## References


1. J. Almendros-Jiménez. An Encoding of XQuery in Prolog. In Z. Bellahsène, E. Hunt, M. Rys, and R. Unland, editors, *Database and XML Technologies*, volume 5679 of *Lecture Notes in Computer Science*, pages 145–155. Springer Berlin / Heidelberg, 2009.
2. J. Almendros-Jiménez, R. Caballero, Y. García-Ruiz, and F. Sáenz-Pérez. A Declarative Embedding of XQuery in a Functional-Logic Language. Technical Report SIC-04/11, Facultad de Informática, Universidad Complutense de Madrid, 2011. `http://gpd.sip.ucm.es/rafa/xquery/`.
3. F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1999.
4. M. Benedikt and C. Koch. From XQuery to relational logics. *ACM Trans. Database Syst.*, 34:25:1–25:48, December 2009.
5. R. Caballero, Y. García-Ruiz, and F. Sáenz-Pérez. Integrating XPath with the Functional-Logic Language $\mathcal{TOY}$. In *Proceedings of the 13th international conference on Practical aspects of declarative languages*, PADL'11, pages 145–159, Berlin, Heidelberg, 2011. Springer-Verlag.
6. L. Fegaras. Propagating updates through XML views using lineage tracing. In *IEEE 26th International Conference on Data Engineering (ICDE)*, pages 309–320, march 2010.
7. J. González-Moreno, M. Hortalá-González, F. López-Fraguas, and M. Rodríguez-Artalejo. A rewriting logic for declarative programming. In *Proc. European Symposium on Programming (ESOP'96)*, volume 1058 of *LNCS*, pages 156–172. Springer, 1996.
8. M. Hanus. Curry: An Integrated Functional Logic Language (version 0.8.2 march 28, 2006). Available at: `http://www.informatik.uni-kiel.de/~mh/curry/`, 2003.
9. M. Hanus. Declarative processing of semistructured web data. Technical report 1103, Christian-Albrechts-Universität Kiel, 2011.
10. M. Hanus and F. Steiner. Controlling search in declarative programs. In *Principles of Declarative Programming (Proc. Joint International Symposium PLILP/ALP'98*, pages 374–390. Springer LNCS, 1998.
11. F. J. López-Fraguas and J. S. Hernández. $\mathcal{TOY}$: A Multiparadigm Declarative System. In *RTA '99: Proceedings of the 10th International Conference on Rewriting Techniques and Applications*, pages 244–247, London, UK, 1999. Springer-Verlag.
12. D. Olteanu, H. Meuss, T. Furche, and F. Bry. XPath: Looking Forward. In *Workshop on XML-Based Data Management*, pages 109–127. Springer, 2002.
13. D. Seipel, J. Baumeister, and M. Hopfner. Declaratively Querying and Visualizing Knowledge Bases in XML. In *In Proc. Int. Conf. on Applications of Declarative Programming and Knowledge Management*, pages 140–151. Springer, 2004.
14. E. Shapiro. *Algorithmic Program Debugging*. ACM Distiguished Dissertation. MIT Press, 1982.
15. W3C. Extensible Markup Language (XML), 2007.
16. W3C. XML Path Language (XPath) 2.0, 2007.
17. W3C. XML Query Use Cases, 2007. `http://www.w3.org/TR/xquery-use-cases/`.
18. W3C. XQuery 1.0: An XML Query Language, 2007.
19. W3C. XQuery 1.0 and XPath 2.0 Formal Semantics (Second Edition), 2010. `http://www.w3.org/TR/xquery-semantics/`.
20. P. Walmsley. *XQuery*. O'Reilly Media, Inc., 2007.

Germán Vidal (Ed.)

# Logic-Based Program Synthesis and Transformation

21st International Symposium, LOPSTR 2011

Odense, Denmark, July 18–20, 2011

Pre-Proceedings

University of Southern Denmark

# Preface

This book contains the papers presented at the 21st International Symposium on Logic-based Program Synthesis and Transformation, LOPSTR 2011, which is held July 18-20, 2011, co-located with PPDP 2011, the 13th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming, AAIP 2011, the 4th International Workshop on Approaches and Applications of Inductive Programming, and WFLP 2011, the 20th International Workshop on Functional and (Constraint) Logic Programming. Previous LOPSTR symposia were held in Hagenberg (2010), Coimbra (2009), Valencia (2008), Lyngby (2007), Venice (2006 and 1999), London (2005 and 2000), Verona (2004), Uppsala (2003), Madrid (2002), Paphos (2001), Manchester (1998, 1992, and 1991), Leuven (1997), Stockholm (1996), Arnhem (1995), Pisa (1994), and Louvain-la-Neuve (1993). Information about the conference can be found at: `http://users.dsic.upv.es/~lopstr11/`.

The aim of the LOPSTR series is to stimulate and promote international research and collaboration in logic-based program development. LOPSTR traditionally solicits contributions, in any language paradigm, in the areas of specification, synthesis, verification, analysis, optimization, specialization, security, certification, applications and tools, program/model manipulation, and transformational techniques. LOPSTR has a reputation for being a lively, friendly forum for presenting and discussing work in progress. Formal proceedings are produced only after the symposium so that authors can incorporate this feedback in the published papers. The LOPSTR 2011 post-proceedings will be published in Springer's Lecture Notes in Computer Science series.

In response to the call for papers, 28 contributions were submitted from 13 different countries. The Programme Committee decided to accept fourteen full papers and four extended abstracts, basing this choice on their scientific quality, originality, and relevance to the symposium. Each paper was reviewed by at least three Program Committee members or external referees. In addition to the 18 contributed papers, this volume includes the abstracts of the invited talks by two outstanding speakers: John Gallagher (Roskilde University, Denmark) and Fritz Henglein (DIKU, University of Copenhagen, Denmark).

I want to thank the Program Committee members, who worked diligently to produce high-quality reviews for the submitted papers, as well as all the external reviewers involved in the paper selection. I am very grateful to the LOPSTR 2011 Symposium Chair, Peter Schneider-Kamp, and the local organizers for the great job they did in preparing the symposium. I would also like to thank Andrei Voronkov for his excellent EasyChair system that automates many of the tasks involved in chairing a conference.

July, 2011
Odense, Denmark

Germán Vidal
Program Chair

# Organization

## Program Chair

| | |
|---|---|
| Germán Vidal | Universitat Politècnica de València, Spain |

## Symposium Chair

| | |
|---|---|
| Peter Schneider-Kamp | University of Southern Denmark, Odense, Denmark |

## Program Committee

| | |
|---|---|
| Elvira Albert | Complutense University of Madrid, Spain |
| Malgorzata Biernacka | University of Wroclaw, Poland |
| Manuel Carro | Technical University of Madrid, Spain |
| Michael Codish | Ben-Gurion University of the Negev, Israel |
| Danny De Schreye | K.U. Leuven, Belgium |
| Maribel Fernandez | King's College London, UK |
| Raúl Gutiérrez | University of Illinois at Urbana-Champaign, USA |
| Mark Harman | University College London, UK |
| Frank Huch | C.A.U. Kiel, Germany |
| Michael Leuschel | University of Düsseldorf, Germany |
| Yanhong Annie Liu | State University of New York at Stony Brook, USA |
| Kazutaka Matsuda | Tohoku University, Japan |
| Fred Mesnard | Université de La Réunion, France |
| Ulrich Neumerkel | Technische Universität Wien, Germany |
| Alberto Pettorossi | University of Roma Tor Vergata, Italy |
| Carla Piazza | University of Udine, Italy |
| Peter Schneider-Kamp | University of Southern Denmark, Denmark |
| Hirohisa Seki | Nagoya Institute of Technology, Japan |
| Josep Silva | Universitat Politècnica de València, Spain |
| German Vidal | Universitat Politècnica de València, Spain |
| Jurgen Vinju | Centrum Wiskunde & Informatica, The Netherlands |
| Jianjun Zhao | Shanghai Jiao Tong University, China |

## Additional Reviewers

Puri Arenas
Jonathan Brandvein
Iliano Cervesato
Agostino Dovier
Fabio Fioravanti
Andrea Formisano
Ángel Herranz
Anil Karna
Matthew Lakin
Dacheng Lei
Manuel Montenegro
Ginés Moreno
Naoki Nishida
Paolo Pilozzi
Valerio Senni
Thomas Stroeder
Dean Voets
Shin Yoo
Carl Friedrich Bolz
Alberto Casagrande
Xi Chang
Santiago Escobar
Marc Fontaine
Samir Genaim
Dragan Ivanovic
Vitaly Lagoon
Julia Lawall
Bo Lin
José F. Morales
Olivier Namet
Susumu Nishimura
Maurizio Proietti
Jon Sneyers
Salvador Tamarit
Jan Wielemaker
Vadim Zaytsev

## Sponsors

University of Southern Denmark

Danish Agency for Science, Technology and Innovation



## Table of Contents

# Program Analysis With Regular Tree Languages*

John P. Gallagher

Computer Science, Building 43.2, Universitetsvej 1,
Roskilde University, DK-4000 Denmark
Email: `jpg@ruc.dk`

**Abstract.** Finite Tree Automata (FTAs) are mathematical "machines" that define recognisable sets of tree-structured terms; they provide a foundation for describing a variety of computational structures such as data, computation states and computation traces. The field of finite tree automata attracts growing attention from researchers in automatic program analysis and verification; there have been promising applications using FTAs in program specialisation, data flow analysis for a variety of languages, term-rewriting systems, shape analysis of pointer-based data structures, polymorphic type inference, binding time analysis, termination analysis, infinite state model checking and cryptographic protocol analysis. The study of FTAs originated in formal language theory and logic, but unlike string automata, which have been widely applied in computer science, especially in compiler theory, tree automata have not yet entered the mainstream of computing theory. In this talk, frameworks for designing static analyses based on tree automata are outlined. They can be used both prescriptively, for expressing and checking intended properties of programs, or descriptively, for capturing approximations of the actual states arising from computations. Computational techniques for handling FTAs efficiently are covered. Finally we look at extensions that go beyond the expressiveness of tree automata, as well as integrating arithmetic constraints.

* Work supported by the Danish Natural Science Research Council project *SAFT: Static Analysis Using Finite Tree Automata.*