Acta Univ. Sapientiae, Informatica, **1**, 2 (2009) 165–213

Extended structural recursion and XSLT

Balázs Kósa
Eötvös Loránd University
Faculty of Informatics
Department of Information Systems
email: balhal@inf.elte.hu

András Benczúr
Eötvös Loránd University
Faculty of Informatics
Department of Information Systems
email: abenczur@inf.elte.hu

Attila Kiss
Eötvös Loránd University, Faculty of Informatics
Department of Information Systems
email: kiss@inf.elte.hu

Abstract. In this paper we describe a simulation of a practically important fragment of XPath 1.0 [16] and XSLT 1.0 [17] with extended structural recursions, which in turn immediately offers us a top-down implementation strategy working in time $O(|D|^2|Q|)$. Here, $|D|, |Q|$ respectively denote the size of the data and the query. However, if the size of the variables is restricted with a constant, then the evaluation works in $O(|D||Q|)$ time. Structural recursions are insensitive to the order of the edges (in our XML model instead of nodes, edges represent elements); hence, in this respect, they are of a weaker expressive power than the more usual models of XML query languages [14]. Still, a large fragment of the most frequent scenarios appearing in practice can be captured with them, which underlines their importance.

1 Introduction

Structural recursion is a graph traversing and restructuring operation applied in many fields of computer science including syntax analysis, code generation

AMS 2000 subject classifications: 68U99
CR Categories and Descriptors: H.2.3 [Database Management]: Languages – XPath, XSLT; D.3.3 [Programming Languages]: Language Constructs and Features – Recursion
Key words and phrases: XML, XPath, XSLT, structural recursion

and program transformation. In the context of databases it was already recommended as a query language alternative in the early 90's to be able to overstep the limitations of the relational data model [6]. The rising of semistructured databases and XML [15] put structural recursions again in the limelight. It formed the basis of UnQL [7] and the core of XSLT [17], where in each step the children of the current node are selected to be processed in the next step. In [11] structural recursions were examined in the context of the typechecking problem. However, in all of these works only a simpler version of the operation was considered.

In [2] we offered a new way of defining the semantics using a special kind of intersection similar to its counterpart in automata theory. We also showed how this new approach intertwines with simple "typing systems" of semistructured data, where simulations are used to prescribe the structure of the data. In [3] we introduced not-isempty conditions in *if... then... else...* statements to be able to define different behaviours depending on the underlying subtree of the processed edge. We analyzed the complexity of the satisfiability and containment problem of such structural recursions.

In this paper going further we extend our model with registers, with which the results of different structural functions called on the same data fragment are connected (structural recursions consist of structural functions). To underpin the usefulness of this extension, we simulate a fragment of XPath 1.0 [16] (XPath_0) and XSLT 1.0 (XSLT_0 [5]).

In XPath_0 only the use of location paths with predicates is supported (i.e. there are no arithmetical or string operations). In the predicates the nonemptiness of such paths, the equality of their results with a constant using existential semantics can be checked and the Boolean combinations of such conditions can be taken. The simulation immediately offers us an implementation strategy in worst case working in time $O(|D|^2|Q|)$. However, if the size of variables, i.e., the size of the list of edges which the variable is equal to, is restricted with a constant, then the implementation works in time $O(|D||Q|)$ both for XPath_0 and XSLT_0. This means that our approach has the same efficiency as the method developed by Gottlob et al. in [8]. Here, $|D|, |Q|$ respectively denote the size of the data and query.

In this paper, we only consider axes `child, parent, ancestor`, `descendant`, but it is not difficult to extend the model to be able to handle the rest of the axes. We process XML trees in a top-down manner and we argue that our approach only processes those elements that are inevitably processed by such an evaluation strategy.

Structural recursions are insensitive to the order of the edges (in our XML

model instead of nodes, edges represent elements), hence in this respect they are of a weaker expressive power than the more usual models of XML query languages [14]. Still, a large fragment of the most frequent scenarios appearing in practice can be captured with them, which underlines their importance. For example in [5] three important features of XSLT_0 are highlighted, which are very useful in practical applications. Firstly, one can use variables to "look forward" in the document. Secondly, variables can be passed as parameters between templates. Thirdly, using modes one can process the same data fragment with different templates.

In Section 2 we introduce our data model, XPath_0 and XSLT_0. In the latter two descriptions we heavily rely on the results of [5, 9]. In Section 3 the syntax and semantics of structural recursions are given. The rewriting method of XPath_0 is presented in Section 4, while in Section 5 XSLT_0 is modelled.

2 Preliminaries

Data model

We consider XML documents as rooted, ordered, directed, unranked trees. However, in our setting, since it is more natural to define structural recursions in edge-labelled trees, we assume that instead of nodes, edges with labels represent tags. It is very easy to rewrite a node-labelled tree into an edge-labelled one and vice versa [1]. In the sequel we refer to these trees as *document trees*. In accordance with [15], we also assume that each document tree has a distinguished document edge, the only outgoing edge of the root, with label /. Furthermore, the document edge is followed by the root edge representing the first element of the corresponding document. The document tree of XML document

```
<a><b>xy</b><c>wz</c></a>
```

can be found in Fig. 2(a).

Formally, we introduce three constructors: the empty tree {} consisting of a node only, the singleton set $\{\mathtt{l} : \mathtt{t}\}$, which is a directed $\mathtt{l}$ edge with subtree $\mathtt{t}$ in its end node, and the append operation @. In $\mathtt{t}_1@\mathtt{t}_2$ the roots of $\mathtt{t}_1$ and $\mathtt{t}_2$ are pulled together. It is not difficult to see that by using these constructors every document tree can be built up [7]. For example $\{\mathtt{a} : \{\mathtt{c} : \{\}\}@\{\mathtt{d} : \{\}\}\}@\{\mathtt{b} : \{\mathtt{e} : \{\}\}\}$ stands for the tree of Fig. 2(b). Furthermore, this construction also gives us a notation to represent document trees. These representations are said to be *ssd-expressions* [1] (ssd: semistructured data). An edge $\mathtt{e}_1$ precedes $\mathtt{e}_2$ if

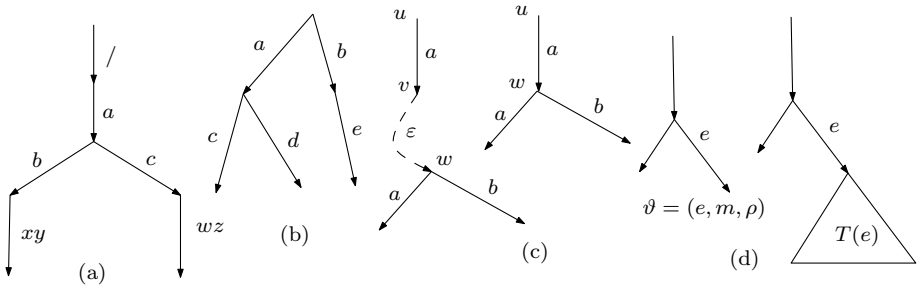

Figure 1: (a) The document tree of the XML document on Page 167. (b) The tree represented by the ssd-expression on Page 167. (c) Elimination of an ε edge. (d) Substitution of a local configuration with the appropriate subtree.

in the corresponding ssd-expression e_1 is written first. Note that this order corresponds to the document order of an XML document [15].

Now, let Σ be a finite alphabet. In what follows, we denote $\mathcal{T}^{\Sigma}$ the set of document trees with edge-labels from Σ. A finite sequence of such trees: $t_1, \ldots, t_m$ constitutes a forest, their set is denoted $\mathcal{F}^{\Sigma}$. With $\mathcal{T}^{\Sigma}(B), \mathcal{F}^{\Sigma}(B)$ we denote that the leaves are labelled with labels from set B. In the following, for graph G we denote with $V.G, E.G$ its node and edge set respectively. For a document tree t $\mathtt{lab}(u)$ we will give the label of node u, $\mathtt{lab} : E.t \to \Sigma$. On the other hand $T : \Sigma \cup \{*\} \to E$ ($E \subseteq E.t$) is defined s.t. $T(\sigma) = \{e \mid e \in E.t \wedge \mathtt{lab}(e) = \sigma\}$ and $T(*) = E.t$.

We are to concentrate mainly on the graph traversing nature of XPath and XSLT, hence, except for the document edge, we suppose that all edges are of the same type. This means that we do not deal with attribute, processing instruction edges, etc. [16]. Our methods can be extended in a straightforward manner to handle edges with types. Furthermore, we assume that every edge has an associated value. Again, we suppose that these values are of the same type, and in contrast to [15] we do not specify how they are obtained. We denote their recursively enumerable set with $\mathcal{D}$. For a given edge in t, function $\mathtt{val} : E.t \to \mathcal{D}$ gives its associated value. In accordance with [18], we assume that in a document tree every edge e has a unique identifier, $\mathtt{id}(e)$ in notation.

For intermediate results of the constructions we shall consider forests from $\mathcal{F}^{\Delta \cup \varepsilon}$. Here, Δ denotes the set of output symbols. The role of an ε edge will be similar to the role of silent transitions in automata, and they will be eliminated similarly. Formally, for edges $(u, a, v), (v, \varepsilon, w)$ in tree t, an (u, a, w) edge is added to t, and the former two edges are deleted (cf. Fig. 2(c)).

e: Expression, p: Predicate, χ: Axis	
$e ::=$	$\chi :: \tau[p_1] \dots [p_n] \mid e_1/e_2$
$\chi ::=$	child | descendant | parent | ancestor | self
$p ::=$	$e \mid e = d \mid e = \$X \mid (p_1 \wedge p_2) \mid (p_1 \vee p_2) \mid \neg(p) \mid [p]$
$d \in \mathcal{D}, X \in \mathcal{V}, \tau \in \Sigma \cup \{*\}$	

ε: Expression$\rightarrow (\mathsf{E.t} \rightarrow 2^{\mathsf{E.t}})$, ε^p :Predicate$\rightarrow (\mathsf{E.t} \rightarrow \{\mathtt{true}, \mathtt{false}\})$, $\rho : \mathcal{V} \rightarrow 2^{\mathsf{E.t}}$, where $\rho(x)$ is a finite set, $\mathcal{V}$ is the set of variables	
$\varepsilon[\![\chi :: \tau[p_1] \dots [p_n]]\!](x) =$	$\{y \mid x\chi y \wedge y \in \mathsf{T}(\tau) \wedge \varepsilon^p[\![p_1]\!](y) = \mathtt{true} \wedge \dots \wedge \varepsilon^p[\![p_n]\!](y) = \mathtt{true}\}$
$\varepsilon[\![e_1/e_2]\!](x) =$	$\cup_{y \in \varepsilon[\![e_1]\!](x)} \varepsilon[\![e_2]\!](y)$
$\varepsilon^p[\![e]\!](x) =$	true iff $\varepsilon[\![e]\!](x)$ is not empty
$\varepsilon^p[\![e = d]\!](x) =$	true iff $\exists y, y \in \varepsilon[\![e]\!](x) \wedge \mathtt{val}(y) = d$
$\varepsilon^p[\![e = X]\!](x) =$	true iff $\exists y, z$ s.t. $y \in \varepsilon[\![e]\!](x) \wedge z \in \rho(X) \wedge \mathtt{val}(y) = \mathtt{val}(z)$
$\varepsilon^p[\![(p_1 \theta p_2)]\!](x) =$	true iff $\varepsilon^p[\![p_1]\!](x) \theta\, \varepsilon^p[\![p_2]\!](x)$ is true $\theta \in \{\wedge, \vee\}$
$\varepsilon^p[\![\neg(p)]\!](x) =$	true iff $\varepsilon^p[\![p]\!](x)$ is false
$\varepsilon^p[\![[p]]\!](x) =$	true iff $\varepsilon^p[\![p]\!](x)$ is true
$x, y, z \in \mathsf{E.t}, d \in \mathcal{D}, X \in \mathcal{V}$	

Figure 2: The syntax (first table) and semantics (second table) of XPath_0.

XPath_0

XPath has already grown to be a widely known and applied language, thus we restrict ourselves to give only the syntax and semantics rules of the fragment we are going to examine (Fig. 2) with a short explanation. For a more formal and exhaustive presentation, consider [9].

We assume that axis names: `child`, `descendant`, `parent`, `ancestor`, `self` are self-describing. The basic building blocks of XPath expression are location steps: $\chi :: \tau[p_1] \dots [p_n]$. Here, χ is an axis, τ is called *edge test* (node test in [16]), which is from $\Sigma \cup \{*\}$, while p_i-s are predicates that are used to filter the returned set of edges. As an example, consider location step

```
child::a[child::b=5],
```

which returns those `a` children of the actual edge that have a `b` child.

In [9, 16] the semantics are given in terms of *contexts*. A context consists of a *context-edge*, a *context-position* and a *context-size*. However, here, owing to the restricted use of functions, e.g. in XPath_0 we do not use functions

`position()`, `first()` and `last()`, it is enough to consider the context-edge or *actual edge*, as it is frequently called. First, we assume there is a given edge on which the evaluation of the expression starts. This edge is usually determined by the host language. Then each location step selects a set of edges, which in turn serve as starting edges of the next location step. This mechanism is explained formally in Fig. 2 (second table).

Note the use of a variable assignment in the semantics. Usually, this is also given by the host environment. Here, $\mathcal{V}$ denotes the recursively enumerable set of variables. In accordance with the syntax of XPath instead of X we refer to a variable as $X. Finally, note also the use of brackets in predicates. These may be omitted, when the precedence rules among Boolean operators give the same evaluation order.

XSLT$_0$

Again, we do not explain XSLT in detail, but rather give an informal overview and a concise formal definition of the semantics. For a more detailed explanation, consider [5]. There, three important features of XSLT$_0$ are highlighted. Firstly, by means of variables one can "look forward" in the document. Secondly, variables can be passed as parameters between templates. Thirdly, using nodes one can process the same data fragment with different templates. All of these properties appear in the XSLT program of Fig. 3, which is called on XML documents of Fig. 4. The program firstly selects the `id`-s of those groups whose top manager is John. Then it selects the `id`-s of the "subgroups" of the former groups, in which an employee with name `Ann` works.

In T_1 we store in variable `X` the `id`-s of those groups that have an employee with name `Ann`. In T_2 groups with John as top manager are selected. Finally, in T_3 using variable `X` subgroups having an employee named `Ann` are chosen. Note that in the XPath expressions we have used the abbreviated syntax [16]. Rewriting the XPath$_0$ expressions with the syntax of Fig. 2 is straightforward.

An XSLT program consists of templates, i.e., `xsl:template` elements. At a given step, we assume that there is a list of edges, E, which has been chosen at the former step. At the beginning of the evaluation this set consists of the document edge of the document. We process the edges of E in document order. Suppose that the actual edge is e. First, we select that template, which fits e, i.e., e satisfies the condition given by the XPath expression of the `match` attribute of the `xsl:template` element. According to the specification of [17], this template should be unambiguous. Then e is processed by this template (it serves as the actual edge for the XPath$_0$ expressions), and another list of

```
<xsl:template match="/">                                    (T1)
  <xsl:variable name="X" select="//group[emp/name='Ann']/id"/>
  <result>
    <xsl:apply-templates select="//group" mode="top">
      <xsl:with-param name="X" select="$X"/>
    </xsl:apply-templates>
  </result>
</xsl:template>

<xsl:template match="group" mode="top">                     (T2)
  <xsl:param name="X"/>
  <xsl:if test="topMgr/name='John'">
    <topGroup>
      <id>
        <xsl:value-of select="id"/>
      </id>
    </topGroup>
    <xsl:apply-templates select="//group" mode="Ann">
      <xsl:with-param name="X" select="$X"/>
    </xsl:apply-templates>
  </xsl:if>
</xsl:template>

<xsl:template match="group" mode="Ann">                     (T3)
  <xsl:param name="X"/>
  <xsl:if test="id=$X">
    <id>
      <xsl:value-of select="id"/>
    </id>
  </xsl:if>
</xsl:template>
```

Figure 3: An XSLT program.

edges is selected by the `xsl:apply-template` element for further processing. When these edges are all processed, then the edge after e in E is considered. Note that in our example T_2 and T_3 may be called on the same elements.

Syntax. For the formal definition of semantics we use a more abstract representation of an XSLT$_0$ program [5]. A template is formalized as an (m, σ)-*rule* as follows:

```
<groups>
  <group>
    <id>G01</id>
    <name>sales</name>
    <topMgr>
      <id>03</id>
      <name>John</name>
    </topMgr>
    <emp>
      <id>04</id>
      <name>Tom</name>
    </emp>...
    <emp>...</emp>
    <group>
      <id>G02</id>
      <name>PR</name>
      <emp>
        <id>05</id>
        <name>Steven</name>
      </emp>
    </group>...
    <group>...</group>
  </group>
</groups>
```

Figure 4: A fragment of an XML document.

$$
\begin{aligned}
&\textit{template}\ m(\sigma, x_1, \ldots, x_n)\\
&\quad \textit{vardef}\\
&\qquad y_1 := r_1; \ldots ; y_s := r_s;\\
&\quad \textit{return}\\
&\qquad \text{if } c_1 \text{ then } z_1; \ldots \text{ if } c_k \text{ then } z_k; \text{ else } z_{k+1};\\
&\textit{end}.
\end{aligned}
$$

Here, m is a mode, $\sigma \in \Sigma$, the latter element shows that the template is called on σ elements (in our data model on σ edges). As it has been shown in [5] we may suppose that only XPath expressions comprising a single element name appear as values of `match` attributes of `xsl:template` elements. σ gives

```
<xs:schema>
  <xs:element name="groups" type="groupsType"/>
  <xs:element name="group" type="groupType"/>
  <xs:element name="topMgr" type="empType"/>
  <xs:element name="emp" type="empType"/>
  <xs:element name="name" type="xs:string"/>
  <xs:element name="id" type="xs:string"/>
  <xs:complexType name="groupsType">
    <xs:sequence>
      <xs:element ref="group" minOccurs="1" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="groupType">
    <xs:sequence>
      <xs:element maxOccurs="1" minOccurs="1" ref="id"/>
      <xs:element maxOccurs="1" minOccurs="1" ref="name"/>
      <xs:element maxOccurs="1" minOccurs="0" ref="topMgr"/>
      <xs:element maxOccurs="unbounded" minOccurs="1" ref="emp"/>
      <xs:element maxOccurs="unbounded" minOccurs="0" ref="group"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="empType">
    <xs:sequence>
      <xs:element maxOccurs="1" minOccurs="1" ref="id"/>
      <xs:element maxOccurs="1" minOccurs="1" ref="name"/>
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

Figure 5: The XML Schema which the XML document fragment of Fig. 4 conforms to.

this element name. $x_1, \ldots, x_n$ are the parameters, while $y_1, \ldots, y_s$ are local variables. r_i is either a constant in $\mathcal{D}$, or it is an XPath_0 expression ($1 \leq i \leq s$). Here, y_j may appear in r_i, if $j < i$. c_j-s are conditions that are either constants ($\mathtt{true}, \mathtt{false}$), or Boolean combinations of atomic conditions: $x = d$, e, $e = d$, $e = x$ ($1 \leq j \leq k, x \in \mathcal{V}, d \in \mathcal{D}, e$ is an XPath_0 expression). Note that all of these tests may appear in XPath_0 expressions (Fig. 2), and their meaning is the same as there. For instance the single condition e becomes true iff e

results a non-empty edge set. Finally, z_s-s are all forests in $\mathcal{F}^{\Delta}(\mathcal{AT})$, i.e., their edge-labels are from Δ (the alphabet of output symbols), and the leaves may be labelled with elements of $\mathcal{AT}$, where $\mathcal{AT}$ denotes the set of *apply-template*-, or shortly *at-expressions*.

An at-expression is of the form $m(p, \bar{x}, \rho)$, where m is mode, p is an XPath_0 expression, with which the next list of edges is selected for processing; $\bar{x}$ is the set of variables that are passed as parameters to the next instantiated template, and $\rho : \mathcal{V} \to 2^{E.t}$ is a variable assignment.

Example 1 *Consider the rewriting of the XSLT$_0$ program of Fig. 3.*

```
template st(/, ε)                                   (T1)
    vardef
        x :=//group[emp/name='Ann']/id;
    return
        if true then {result : {}}; at-expr: top(//group, x, ρ)
end.

template top(group, x)                              (T2)
    return
        if topMgr/name='John' then {topGroup : {id : {}}@{}};
            at-expr1: val(id, ε, ε), at-expr2: Ann(//group, x, ρ)
end.

template Ann(group, x)                              (T3)
    return
        if id= x then {id : {}}; at-expr1: val(id, ε, ε)
end.

template val(id, ε)                                 (T4)
    return
        if true then {val(id) : {}};
end.
```

Here, T_1 is called on the document edge /, in mode st with no parameters. z_1 consists of a single node, whose at-expression label is top(//group, x, ρ), where ρ assigns to x the result of the XPath expression of the vardef part. We return to the special role of the (st, /)-rule soon. In T_2 the at-expressions after at-expr1, at-expr2 respectively belong to the leaf of the id edge and the

leaf appended to this edge. Note the superficial nature of this construction. Furthermore, the `xsl:value-of` element returning the value of `id` children is represented here as a special template T_4 also returning the value of `id` elements. The at-expression of T_3 invokes T_4 on the `id` children of the actual edge. It is easy to verify that with this rewriting we simulate correctly the functioning of the former `xsl:value-of` element.

As we have already mentioned, according to [17], the conflict of templates should be avoided, hence only one (m, σ)-rule is allowed to give for each (m, σ)-pair. Furthermore, we assume that there is a distinguished *start mode*, $\mathtt{st}$, which is only used with the document edge. For sake of simplicity, we also assume that $(\mathtt{st}, /)$ is the only rule, where variable definitions are allowed to give. It has only one condition, where c_1 is $\mathtt{true}$, and z_1 constructs a $\mathtt{result}$ edge as the root edge of the output. (With this, we guarantee that the output is always a tree.) In the XSLT program of Fig. 3 (T_1) stands for this rule. Note that according to [17] it is not allowed to be given a mode to a template instantiated on the document edge. Nevertheless, this slight deviation carries no importance.

Semantics. A run of a program on t is given in terms of trees in $\mathcal{T}^{\Delta}(\mathsf{LC}(t)^*)$. Here, $\mathsf{LC}(t) = \mathsf{E}.t \times M \times \Psi$ denotes the set of *local configurations*, where M is the set of modes and Ψ is the set of partial variable assignments from $\mathcal{V}$ to $2^{\mathsf{E}.t}$ with a finite domain. $(e, m, \rho) = \vartheta \in \mathsf{LC}(t)$ represents that case, when (m, σ)-rule T is to be applied on e, where $\mathtt{lab}(e) = \sigma$, and the parameters of T are in the domain of ρ. Here, e is the edge whose endnode is labelled with ϑ. For a tree in $\mathcal{T}^{\Delta}(\mathsf{LC}(t)^*)$, a leaf may be labelled with a sequence of local configurations.

Informally, suppose that a subtree of the result $\hat{t} \in \mathcal{T}^{\Delta}(\mathsf{LC}(t)^*)$ has been already constructed. For sake of transparency we consider a leaf labelled with a single local configuration $\vartheta = (e, m, \rho)$. This defines an (m, σ)-rule T to be applied. Here, e will be used as the actual edge of the XPath_0 expressions of T. When $\mathsf{T}(e)$ is evaluated, the result (in $\mathcal{F}^{\Delta}(\mathsf{LC}(t)^*)$) should be added to the leaf and ϑ should be deleted. If the leaf is labelled with a sequence containing more local configurations, the former method should be applied to each of them consecutively (cf. Fig. 2(d)).

The process starts with t_0, where t_0 consists of an edge with label /, and with leaf label $(/, \mathtt{st}, \epsilon)$ (this guarantees that the $(\mathtt{st}, /)$-rule is called first), and it stops, when there is no $\mathsf{LC}(t)^*$ label in the constructed output to process further. We shall assume that, if (m, σ)-rule T has been already applied to an edge e, then T is not allowed to be applied on e again. Thus, we avoid infinite loops. This feature is guaranteed when XSLT_0 programs are simulated with

structural recursions.

Formally, an XSLT$_0$ program is a tuple $P = (\Sigma, \Delta, M, st, R)$, where Σ and Δ are the labeling alphabets of the input and output respectively. M is the set of modes disjunct from Σ and Δ. st is the start mode and R is a finite set of (m, σ)-rules.

We define a *rewrite relation induced by* P *on* t, $\rightarrow_{P,t}$. Here

$$\rightarrow_{P,t}: \mathcal{T}^{\Delta}(LC(t)^*) \rightarrow \mathcal{T}^{\Delta}(LC(t)^*).$$

For $\xi, \zeta \in \mathcal{T}^{\Delta}(LC(t)^*)$, first we explain the simpler case, when a leaf labelled with a single local configuration is considered. Then $\xi \rightarrow_{P,t} \zeta$, if ξ has a leaf edge e with leaf label (e, m, ρ), where $lab(e) = \sigma$, and the parameters of (m, σ)-rule T are all included in the domain of ρ. ζ is constructed by substituting e with f_o, the result of T instantiated on e with variable assignment ρ.

To understand the construction of f_o, remember the syntax of T. First we evaluate r_i-s taking e as the actual edge to get the possible values of y_i-s ($1 \leq i \leq s$). Denote E_i the result of r_i. Then, assuming that c_j is the first condition becoming true, $z_j \in F^{\Delta}(\mathcal{AT})$ is transformed into f_o, where we substitute each at-expression leaf label with a sequence of local configurations. Namely, a leaf label $m'(p, \bar{z}, \rho')$ is substituted with sequence $(e_1, m', \rho'), \ldots, (e_l, m', \rho')$. Here $\bar{z} \subseteq \{x_1, \ldots x_n, y_1, \ldots y_m\}$, $\rho'(x_i) = \rho(x_i)$ and $\rho'(y_j) = E_j$, $p(e) = \{e_1, \ldots, e_l\}$ (p is the XPath$_0$ expression of the at-expression in question, while e is the actual edge on which T is instantiated). Furthermore, e_r precedes e_o in document order, if $r < o$ ($1 \leq i \leq n, 1 \leq j \leq s, 1 \leq r, o \leq l$).

If e is labelled with a sequence local configurations, then ζ should be obtained by applying the former method to each local configuration one after the other.

The initial local configuration is defined to be t_0. With this the transformation realized by P, is the (partial) function $\tau_P : \mathcal{T}^{\Sigma} \rightarrow \mathcal{T}^{\Delta}$, with $\tau_P(t) = s$, if $t \in T^{\Sigma}$, $s \in T^{\Delta}$, and $t_0 \rightarrow^*_{P,t} s$.

3 Structural recursions

Syntax

A structural recursion f is constituted by *structural functions*, in notation $f = (f_1, \ldots, f_n)$, which can call each other. In the definition of a structural function we consider inputs given with ssd-expressions and specify what should happen for the different constructors. For the syntax of a row of a structural

SR: structural recursion, Reg: registers, $r(f_i)$: a row of f_i, C: condition, C^d: condition for the default, $Sf = \{f_1(t), \ldots, f_n(t)\}$, $a \in \Sigma, \alpha \in \mathcal{D}, y \in \mathcal{V}, t \in \mathcal{T}^\Sigma$, n.i. stands for *not isempty*	
$SR ::=$	S, Reg
$S ::=$	$f_i : r(f_i)$, $(1 \leq i \leq n)$
$r(f_i) ::=$	$(t_1@t_2) = f_i(t_1)@f_i(t_2) \mid (\{\}) = \{\} \mid (\{a : t\}) = R \mid$ $(\{* : t\}) = R^d \mid$ $(\{a : t\}) =$ if C_1 then R_1 else $R_2 \mid$ $(\{* : t\}) =$ if C_1^d then R_1^d; else R_2^d
$R ::=$	fo, where $fo \in \mathcal{F}^{\Delta}(Sf)$
$R^d ::=$	fo, where $fo \in \mathcal{F}^{\Delta \cup \{*\}}(Sf)$
$C ::=$	$n.i.(f_j(t)) \mid val(a) = \alpha \mid val(a) = y \mid (C_1 \wedge C_2) \mid$ $(C_1 \vee C_2) \mid \neg(C)$
$C^d ::=$	$n.i.(f_j(t)) \mid val(*) = \alpha \mid val(*) = y \mid (C_1 \wedge C_2) \mid$ $(C_1 \vee C_2) \mid \neg(C)$
$Reg ::=$	$X^b_{f_i} = X^b_{f_j} \mid (Reg_1 \wedge Reg_2) \mid (Reg_1 \vee Reg_2) \mid \neg(Reg)$

$\varepsilon^C : condition \to (E.t \to \{true, false\})$, $\theta \in \{a, *\}$, $\rho : \mathcal{V} \to 2^{\mathcal{D}}$	
$\varepsilon^p[\![val(\theta) = \alpha]\!](e) =$	true iff $lab(e) = \theta, val(e) = \alpha$
$\varepsilon^p[\![val(\theta) = y]\!](e) =$	true iff $lab(e) = \theta, \exists e' \in \rho(y)$ s.t. $val(e) = val(e')$

Figure 6: The syntax rules of structural recursion $f = (f_1, \ldots, f_n)$ (first table). The semantics of a subset of conditions (second table).

function, consider the $r(f_i)$ row of the first table of Fig. 6. Here, it turns out that for constructors $t_1@t_2, \{\}$ structural functions always work in the same manner, hence they are not given in the definition. For $t_1@t_2$ they call themselves both on t_1 and t_2, and at the end append the results. For $\{\}$, they construct a single node.

Example 2 *As an example, we give structural recursion* $f = (f_1, f_2, f_3)$*, which copies a subtree* $\{a : t\}$ *if the* a *edge has an* Ann *child.*

f_1: $(\{a : t\}) =$ *if n.i.*$(f_2(t))$ *then* $\{a : f_3(t)\}$
$(\{l : t\}) = f_1(t)$

f_2: $(\{Ann : t\}) = \{\psi : \{\}\}$
$(\{l : t\}) = \{\}$

f_3: $(\{* : t\}) = \{* : f_3(t)\}$

Example 3 *As another example we give the rewriting of XPath$_0$ expression:*

$$q_1 = \texttt{self::a[par::c=5]/child::b/desc::d/par::a},$$

where `desc, par` *are abbreviations of axes* `descendant, parent` *respectively.*

f_{a_1}: $(\{a : t\}) = f_b(t)$
$(\{* : t\}) = f_{a_1}(t)$

f_b: $(\{b : t\}) = f^1_d(t)$
$(\{* : t\}) = f_{a_1}(t)$

f^1_d: $(\{d : t\}) = \{\psi : \{\}\}@\, f^1_d(t)$
$(\{* : t\}) = f^1_d(t)$

f_{a_2}: $(\{a : t\}) =$ *if n.i.*$(f^2_d(t))$ *then* $\{\psi : \{\}\}@\, f_{a_2}(t)$ *else* $f_{a_2}(t)$
$(\{* : t\}) = f_{a_2}(t)$

f^2_d: $(\{d : t\}) = \{\psi : \{\}\}$
$(\{* : t\}) = \{\}$

$f^{pr_1}_c$: $(\{c : t\}) =$ *if* $val(c) = 5$ *then* $f^{pr_1}_{a_1}(t)$ *else* $f^{pr_1}_c(t)$
$(\{* : t\}) = f^{pr_1}_c(t)$

$f^{pr_1}_{a_1}$: $(\{a : t\}) = \{\psi : \{\}\}@\, f^{pr_1}_c(t)$
$(\{* : t\}) = f^{pr_1}_c(t)$

$X^d_{fd^1} = X^d_{fd^2}$, $X^a_{f^{pr_1}_{a_1}} = X^a_{f_{a_1}}$

Here, $pr_1 =$ `self::a/par::c=5`. $f^{pr_1}_c$ checks whether the value of the actual c edge is equal to 5, then calls $f^{pr_1}_{a_1}$, which constructs a ψ edge as a result of an a edge, if it immediately follows the aforementioned c edge.

`self::a/child::b/desc::d`, `desc::d/par::a`

are simulated with f_{a_1}, f_b, f^1_d, f_{a_1}, f^2_d, respectively. The connection is given with the register restriction $X^d_{f^1_d} = X^d_{f^2_d}$, whose intended meaning is that the d edges processed by f^1_d should also be processed by f^2_d and vice versa. Thus these d edges both have a b ancestor with an a parent, and an a parent. One may consider $X^d_{f^i_d}$-s as *registers* containing the id-s of the d edges processed by f^i_d $(i = 1, 2)$. Note the similar role of $X^a_{f^{pr_1}_{a_1}} = X^a_{f_{a_1}}$. In f_{a_2} n.i. stands for the *not-isempty* condition. The $\{* : t\}$ rows represent the default cases.

Note that only ψ ($\psi \in \Delta$) edges are constructed here. This is understandable, since we simulate an XPath$_0$ expression. In the output, only ψ edges constructed by f_{a_2} representing the last location step `parent::a` will be considered. Thus we get a star as a result, i.e., a single node with outgoing ψ

edges. If for document tree t and e_0 as context edge q_1 selects $e_1, \ldots, e_s$, then the star consists of s ψ edges, the first constructed as a result of processing e_1 according to the a row of f_{a_2}, the second as processing e_2, etc.

Now, consider the syntax rules in the first table of Fig. 6. Note that one may construct forests in $F^{\Delta}(Sf)$, i.e., the leaves of these forest may be labelled with $f_i(t)$-s. The intended meaning of such a label is that the result of f_i called on t is to be connected to the leaf in question. In the previous example with $f_b(t)$ on the right hand side of a row we denoted a node labelled with $f_b(t)$.

Semantics

Operational graphs. As in [2], in order to define the semantics, we introduce *operational graphs*, which will represent the "relationships" among structural functions. For structural recursion $f = (f_1, \ldots, f_n)$, we denote its operational graph with U_f.

In the construction, for each f_i we assign a node with name f_i $(1 \leq i \leq n)$. The edges of U_f are given with respect to the rows of f_i-s. As a warm-up we consider a simple row:

$$(\{a : t\}) = \{b : f_j(t)\}.$$

Here, we add an $a(x)$ edge from f_i to f_j. The intended meaning is obvious, we represent that as a result of singleton $\{a : t\}$ f_i calls f_j. For

$$(\{a : t\}) = \text{ if n.i.}(f_j(t)) \text{ then } f_k(t) \text{ else } f_l(t),$$

we add an $a(x)$ edge with an additional pr (predicate) label to f_j and two other $a(x)$ edges with labels th, el (then, else) to f_k and f_l respectively. If no structural function is called, we use an additional node w. Formally, for row:

- $(\{\theta : t\}) = fo$ $(fo \in \mathcal{F}^{\Delta \cup \{*\}}(Sf), \theta \in \{a, *\})$:

  $(f_i, p, f_j) \in E.U_f$, if $f_j(t)$ is among the labels of leaves of fo, i.e., f_j is called by f_i as a result of a θ edge. If $f_j(t)$ appears more than once among the labels, still only one (f_i, p, f_j) edge is added. Here, if $\theta = a$, then $p = a(x)$. Otherwise, if $\theta = *$, $p = \neg a_1(x) \wedge \ldots \wedge \neg a_l(x)$, where $a_1, \ldots, a_l$ are the symbols appearing in the singleton sets on the left side before the default case in the definition of f_i. If there are no such rows in the definition, then $p = \top(x)$. $a(x)$ is a predicate symbol, which we always interpret over Σ s.t. $a(x)$ becomes true iff $x = a$, while $\top(x)$ is satisfied by all constants of Σ. As an example, consider the operational graph of (f_{a_1}, f_b, f_d^1) of Example 3 in Fig. 7(b) (the leftmost graph). Here and in the rest of the examples we abbreviate $a(x)$ with a.

If f_o has no leaf labels, i.e., no structural function is called, then an (f_i, p, w) edge is added instead of an (f_i, p, f_j) edge.

- $(\{\theta : t\}) =$ if C then fo_1 else fo_2:

  (f_i, p, f_j) is in $E.U_f$ with an additional label pr, if n.i.$(f_j(t))$ appears in C. Such an edge will be called *premise*. The meaning of θ and p is the same as in the previous case.

  Furthermore, (f_i, p, f_k) is in $E.U_f$ with label th if $f_k(t)$ appears among the labels of leaves of fo_1. If $f_k(t)$ appears as a leaf label in fo_2, an el label is added. These edges are called *then-, else-edges* respectively. The premise, then- and else-edges together will be called *conditional edges*. If $f_k(t)$ appears more than once in fo_s, still only one (f_i, p, f_k) edge is added $(s = 1, 2)$. However, if it appears in fo_1 and also in fo_2, then this edge is labelled with both th and el. Again, an (f_i, p, w) edge is added with the appropriate th, el label, if the leaves of fo_s are not labelled. As an example, consider the outgoing a edges of f_{a_2} and the outgoing c edges of $f_c^{pr_1}$ of Example 3 in Fig. 7(b).

An edge of U_f is called *constructor edge*, if it results a construction, i.e., the forest on the right side of the corresponding row is not a single node.

The corresponding operational graph of f_{q_1} in Example 3 can be found in Fig. 7(b). Note that operational graphs are not necessarily connected.

Process of an input. Next, we show how a document tree is processed by an operational graph. For this we introduce a new operation, which is very similar to the intersection operation in automata theory.

Definition 1 *Let* U_f, t *be an operational graph and a document tree* $(f = f_1, \ldots, f_n)$. *Then the* intersection *of* U_f *and* t, *in notation* $U_f \sqcap t$, *is defined as follows:* $V.U_f \sqcap t := \{(\varphi, u) \mid \varphi \in V.U_f, u \in V.t\}$, $(\varphi \in \{f_1, \ldots, f_n, w\})$. $E.U_f \sqcap t := \{((f_i, u), p(x) \wedge a(x), (\varphi, v)) \mid (f_i, p(x), \varphi) \in E.U_f, (u, a, v) \in E.t, \Sigma \models p(a)\}$.

The intuition behind this definition is clear. For instance an $((f_i, u), p \wedge a, (f_j, v))$ edge means that (u, a, v) is processed by f_i, and then f_j is called. Note that we have slightly blurred the distinction between predicates and constants. An edge-label a $(a \in \Sigma)$ of a document tree is considered as predicate $a(x)$. Also note that if $p(x) \wedge a(x)$ is satisfiable, then only $a \in \Sigma$ satisfies it, hence sometimes we shall write a instead of $p(x) \wedge a(x)$ as edge label.

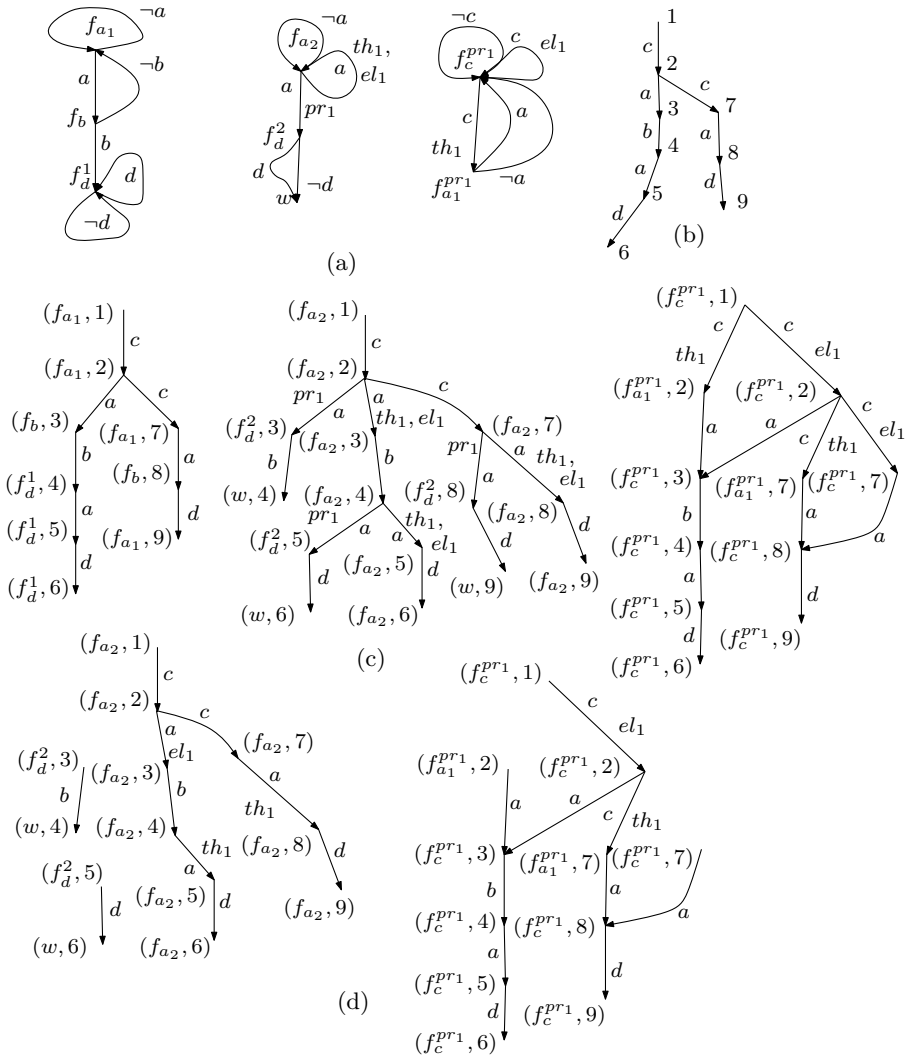

Figure 7: (a) $\mathsf{U}_{\mathsf{f}_{\mathsf{q}_1}}$. (b) An input tree t. (c) $\mathsf{U}_{\mathsf{f}_{\mathsf{q}_1}} \sqcap \mathsf{t}$. (d) A fragment of $\mathsf{G}_{\mathsf{f},\mathsf{t}}$.

In most cases only a subset of structural functions $f_{i_1}, \ldots, f_{i_k}$ is allowed to be called on the document edge. In this case we only consider (f_{i_j}, u_0) as elements of $V.U_f \sqcap t$, where u_0 is the root of t ($1 \leq j \leq k$). Such structural functions will be called *upper structural functions*. In f_{q_1} of Example 3 the upper structural functions are f_{a_1}, f_{a_2} and $f_c^{pr_1}$. An example for intersection can be seen in Fig. 7(c)-(d).

In the sequel, sometimes (e_{U_f}, e_t) will denote an edge in $E.U_f \sqcap t$. Here, e_{U_f}, e_t are called *ancestor images*. (e_{U_f}, e_t) is a *premise* (*constructor*, *then-*, *else-edge*), if its ancestor image in U_f is also a premise (constructor, then-, else-edge).

Deletion of the unnecessary conditional edges. In the next step, in $U_f \sqcap t$ we delete the premises and those then- and else-edges, whose condition is not satisfied. First, we have to note that for

$$Cond = \text{ if } C \text{ then } fo_1 \text{ else } fo_2$$

and for an edge e_t of t, if (e_{U_f}, e_t) is in $E.U_f \sqcap t$ s.t. e_{U_f} is a conditional edge of $Cond$, then e_t form pairs with the rest of the conditional edges of $Cond$. The set of these edges of $U_f \sqcap t$ will be referred as $Cond^{e_t}$. In order to be able to decide whether C is satisfied by e_t and the subtree under e_t, we take a copy of C, which we denote C^{e_t}. The algorithm eliminating the unnecessary conditional edges consists of three steps.

Equality conditions. First, the $val(\theta) = \alpha$ conditions are considered ($\theta \in \Sigma \cup \{*\}$). In Fig. 7(e) we have supposed that $val((1, c, 2)) \neq 5$, hence then-edge $((f_c^{pr_1}, 1), c, (f_{a_1}^{pr_1}, 2))$ is deleted. On the other hand, for $(2, c, 7)$ we have assumed that its value is 5, thus we delete else-edge $((f_c^{pr_1}, 2), c, (f_c^{pr_1}, 7))$.

Formally, in this step, the $val(\theta) = \alpha, val(\theta) = x$ conditions of C^{e_t} are considered. We assume that there is a given variable assignment ρ. With this and the value of e_t we substitute the preceding equality conditions with their truth values (cf. the second table of Fig. 6).

If C^{e_t} becomes true, then clearly, the then-branch should be executed, hence except for the then-edges, we delete all other conditional edges belonging to $Cond^{e_t}$. If an edge is a then- and else-edge at the same time, it is also kept. These and the remaining then-edges are considered as normal (non-conditional) edges in further steps of the algorithm. We refer to this method in the sequel as *deletion with respect to the then-branch*.

On the other hand, if C^{e_t} becomes false, then, for obvious reasons, we delete with respect to the else branch, i.e. edges with label el are kept, and considered as normal edges further on. Note that owing to the presence of n.i. conditions, we may not be able to determine the truth value of C^{e_t}.

The result of this step is denoted $\hat{G}_{f,t}$.

Registers. Second, we consider the constraints given by the registers. For an edge $((f_i, u), b, (\varphi_1, v)) \in \hat{G}_{f,t}$ and a restriction of registers $\mathtt{Reg}$, an atomic condition, $X^b_{f_i} = X^b_{f_j}$, becomes true, if there exists another edge $((f_j, u), b, (\varphi_2, v)) \in \hat{G}_{f,t}$ ($\varphi_1, \varphi_2 \in \{f_1, \ldots, f_n, w\}$). In other words, the b rows of f_i and f_j should be called on (u, b, v). Otherwise, $X^b_{f_i} = X^b_{f_j}$ becomes false. If at the end, $\mathtt{Reg}$ becomes false, then $((f_i, u), b, (\varphi_i, v))$ should be deleted from $\hat{G}_{f,t}$. The result after the evaluation of register restrictions is denoted $\check{G}_{f,t}$. Again, an example can be found in Fig. 7.(e). Here, $X^d_{f^1_d} = X^d_{f^2_d}$ is not satisfied by $((f^2_d, 8), d, (w, 9))$, since $(8, d, 9)$ has not got any b ancestor, f^1_d is not called on it. On the other hand, $((f^1_d, 5), d, (f^1_d, 6))$ and $((f^2_d, 5), d, (w, 6))$ satisfies this register restriction.

Not-isempty conditions. Third, the n.i. conditions are evaluated. (i) If from a premise of $\check{G}_{f,t}$ a constructor edge is reachable through a path not containing any conditional edges, then the n.i. condition in question, n.i.$(f_j(t))$, is satisfied, thus we substitute n.i.$(f_j(t))$ in C^{e_t} with a $\mathtt{true}$ constant. If C^{e_t} becomes true, then we delete with respect to the then-branch. In Fig. 7(e) the n.i. condition of $((f_{a_2}, 4), a, (f^2_d, 5))$ is satisfied.

(ii) If there are neither constructor, nor conditional edges reachable from the premise in question, then there is no further possibility to satisfy the corresponding n.i.$(f_j(t))$ condition, hence we substitute it with constant $\mathtt{false}$. If C^{e_t} becomes false, then we delete with respect to the else-branch. Again, in Fig. 7(e) the n.i. condition represented by premise $((f_{a_2}, 2), a, (f^2_d, 3))$ is not satisfied.

The algorithm stops, when there are no premises left. Suppose now that there are still premises, nevertheless steps (i)-(ii) cannot be applied. This means that each path from a premise to a constructor edge contains at least one conditional edge e beside the premise in question. If e is a then-, or an else-edge, since the equality conditions are all checked, we know that there is also a premise belonging to e. Hence, at the end we conclude that some of the premises form cycles, whereas, clearly, $U_f \sqcap t$ is a tree. Thus, the algorithm surely stops. Denote $G_{f,t}$ the result.

Construction of the result.

Example 4 *To describe the construction of the output we use another structural recursion* $f = (f_1, f_2, f_3)$ *as an example, where* f_1 *is the upper structural function. (The operational graph of* f *can be found in Fig. 8(a).)*

$f_1\colon (\{a : t\}) = \{b : \{\{a : f_2(t)\}@\{c : f_3(t)\}\}\}$ $\quad$ $f_2\colon (\{* : t\}) = \{* : f_2(t)\}$

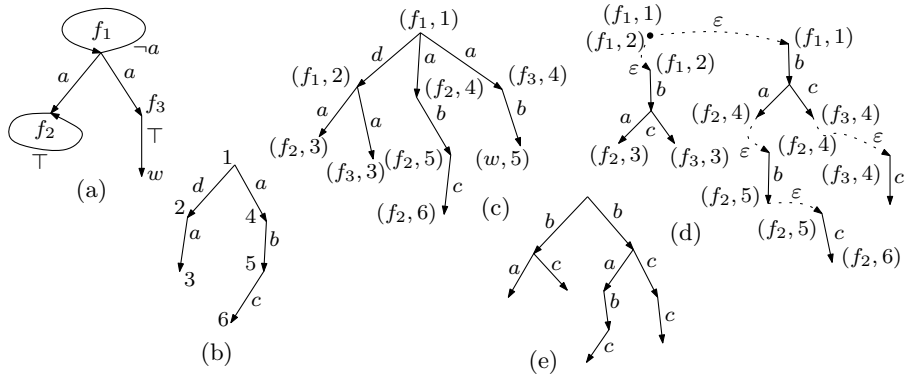

Figure 8: (a) The operational graph of $\mathsf{f} = (\mathsf{f}_1, \mathsf{f}_2, \mathsf{f}_3)$. (b) An input. (c) The intersection of the operational graph of (a) and the input of (b). (d) Construction and connection of the basic forests. (e) The final result.

$$(\{* : \mathsf{t}\}) = \mathsf{f}_1(\mathsf{t}) \qquad\qquad \mathsf{f}_3\colon (\{* : \mathsf{t}\}) = \{\mathsf{c} : \{\}\}$$

If an edge $e_t = (u, a, v)$ of a document tree t is processed according to the a row of f_1, i.e., $((\mathsf{f}_1, u), a, (\mathsf{f}_j, v))$ is in $\mathsf{E}.\mathsf{U}_\mathsf{f} \sqcap \mathsf{t}$ $(j = 1, 2)$, then we take a tree $\hat{\mathsf{t}} = \{\mathsf{b} : \{\{\mathsf{a} : \{\}\}@\{\mathsf{c} : \{\}\}\}\}$, and we label its root with (f_1, u) and the leaves of the b and c edges with (f_2, v), (f_3, v), respectively. The labels of the leaves indicate that the results of f_2 and f_3 applied on the subtrees under e_t should be "connected" to $\hat{\mathsf{t}}$. Similarly, the label of the root shows that which fragment of the result $\hat{\mathsf{t}}$ should be connected to. The connection is accomplished through ε edges. As an example, consider Fig. 8(a)-(e). Note that here, when f_1 is applied to $(1, \mathsf{d}, 2)$, only a node is constructed with two labels (cf. Fig. 8(d)).

Formally, denote $\mathsf{E}(\mathsf{f}_i, e_t)$ the set of neighbouring edges in $\mathsf{G}_{\mathsf{f},\mathsf{t}}$, whose labels are the same, and whose ancestor image is $e_t = (u, a, v)$ in t. Such a set may consists of only one edge. The ancestor images in U_f are all of the form $(\mathsf{f}_i, p, \varphi)$, $\varphi \in \{\mathsf{f}_1, \ldots, \mathsf{f}_n, w\}$, i.e., they belong to the same condition. In accordance with the previous observation, the edges of $\mathsf{E}(\mathsf{f}_i, e_t)$ represent that in the process of t, f_i is called on e_t, as a consequence, forest fo should be constructed, and the structural functions appearing as leaf labels should be called. To represent this, for $\mathsf{E}(\mathsf{f}_i, e_t)$ we take a copy of fo, we substitute the $*$ labels with a (the label of e_t), we label the root with (f_i, u), and if a leaf has label $\mathsf{f}_j(\mathsf{t})$, then we change it to (f_j, v). These new labels will be used to establish the connection between these forests. Denote $\mathsf{fo}(\mathsf{f}_i, e_t)$ the result,

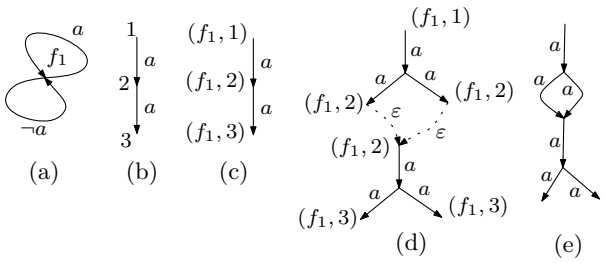

Figure 9: (a) The operational graph of $\mathsf{f} = (\mathsf{f}_1)$. (b) An input. (c) The intersection of the operational graph of (a) and the input of (b). (d) Construction and connection of the basic forests. (e) The final result.

which we call *basic forest* in the sequel.

The output should be constructed from the basic forests. For this, first, add ε edges from the (f_j, ν) leaves to the (f_j, ν) roots. Second, if there are more than one (f_j, ν) roots, then choose one, and add ε edges to the rest of these roots. At the last step the ε edges should be eliminated.

In most cases we get forests as results instead of trees. Hence, one may choose a root structural function f_k among the upper structural functions $(1 \leq k \leq n)$. This means that in the output only the tree reachable from the root of $\mathsf{fo}(\mathsf{f}_k, /)$ (f_k called on the document edge) should be considered.

Example 5 *The next example in Fig. 9(a)-(e) with structural recursion* $\mathsf{f} = (\mathsf{f}_1)$ *shows how our semantics avoids outputs of exponential size.*

$$\begin{aligned}\mathsf{f}_1\colon (\{a : t\}) &= \{a : \{\{a : \mathsf{f}_1(t)\}@\{a : \mathsf{f}_1(t)\}\}\}\\ (\{* : t\}) &= \mathsf{f}_1(t)\end{aligned}$$

Note that the result in Fig. 9(e) does not fit our data model, since it cannot be constructed by using constructors: $@, \{l : t\}, \{\}$. However, it "encodes" all necessary information, and it is easy to unfold it to a tree in our data model.

Order of the result. As a consequence of rule

$$\mathsf{f}_i : (\{t_1@t_2\}) = \mathsf{f}_i(t_1)@\mathsf{f}_i(t_2),$$

for a structural recursion f and document tree t, if e^1 precedes e^2 in document order $(e^1, e^2 \in \mathsf{E}.t)$, then the edges of the fragment constructed by calling f on e^1 and the subtree under e^1 precedes the edges of the fragment

resulted from the process of e^2 and the subtree under e^2. In the simulation of XPath_0 expressions the aforementioned fragments are single ψ edges, hence the document order straightforwardly defines the order of the result.

When XSLT_0 is considered, for each input, the simulated program will define the order among the basic forests.

Number of steps. If we consider the size of f, $|f|$, as the number of equality conditions, structural functions appearing on the right sides of rows (each appearance of f_j counts one) and atomic conditions of registers, and the size of t as $|V.t| + |E.t|$, then the size of $U_f \sqcap t$ is $O(|f||t|)$. The equality constraints can be checked in $O(K|f||t|)$ steps, where $K = \max\{|\rho(x)| \mid x \text{ is used in } f\}$. Clearly, $K \leq |t|$. Note that if there are no variables, then the equality conditions can be checked in $O(|f||t|)$. For the register restrictions, we suppose that for each edge $(u, a, v) = e \in E.t$, we have an array that stores f_i, if $((f_i, u), a, (\varphi, v)) \in \hat{G}_{f,t}$ ($\varphi \in \{f_1, \ldots f_n, w\}, 1 \leq i \leq n$). With this, the register restrictions, satisfied by e, can be found in $O|f|$ steps. Hence register restrictions can be checked in $O(|t||f|)$ steps. Since a path from a premise to a constructor edge contains at most $|f|$ different conditional edges, $G_{f,t}$ can be constructed in $O(|f|^2|t|)$ steps. All in all, supposing that $|f| < |t|$, we get that $f(t)$ can be constructed in $O(|t|^2|f|)$ time. If variables are not used, then $f(t)$ can be constructed in $O(|t||f|^2)$ time. Furthermore, if the number of embedded n.i. conditions is limited with a constant, as in the case of the following simulations of XPath_0 expressions and XSLT_0 programs, then $f(t)$ can be computed in $O(|t||f|)$ time.

4 Rewriting of XPath_0

First of all, we have to define formally the equivalence of an XPath_0 expression and a structural recursion. For this, let

$$q = \chi_1 :: \tau_1[p_{1_1}] \ldots [p_{m_1}]/ \ldots /\chi_n :: \tau_n[p_{1_n}] \ldots [p_{m_n}]$$

be an XPath_0 expression. Then for a document tree t, $e_0 \in E.t$, a sequence of edges $e_0, e_1, \ldots, e_n$ is called a *result-chain (of* q*)*, if $e_i \in E.t$, $e_i \chi_{i+1} e_{i+1}$, $val(e_{i+1}) = \tau_{i+1}$, and $p_{j_{i+1}}(e_{i+1})$ is true ($0 \leq i \leq n-1, 1 \leq j \leq m$). If we denote

$$\chi_1 :: \tau_1[p_{1_1}] \ldots [p_{m_1}]/ \ldots /\chi_j :: \tau_j[p_{1_j}] \ldots [p_{m_j}]$$

with q_{τ_j}, then $e_j \in q_{\tau_j}(e_0)$, i.e., e_j is selected by q_{τ_j} initialized on e_0 ($1 \leq j \leq n$). Furthermore, we say that an edge e is *touched*, if there exists a j s.t. $e \in q_j(e_0)$, or e is touched in one of the pr_{i_j}-s. A touched edge e is *uppermost*,

if there is not any edge $e' \in E.t$ s.t. e' is also touched, and e' is an ancestor of e.

Lemma 1 *Let* q *be an XPath*$_0$ *expression without predicates,* t *a document tree,* $e_0 \in E.t$. *Then, there exists at most one uppermost node for* t *and* e_0.

Proof. Suppose that $q(e_0)$ is not empty, i.e., there exists at least one result chain, and there are two uppermost nodes e and e'. Then neither e is the ancestor of e', nor e' is the ancestor of e. However, since t is a tree, they have at most one common ancestor e''. As we only use axes `self, child, desc, par, anc`, it is easy to see, that e'' is touched, thus we get a contradiction.

If $q(e_0)$ is empty, then let i be that maximal number, to which $q_{\tau_i}(e_0)$ is not empty. Then, the previous reasoning can be applied to q_{τ_i}. If $q_{\tau_i}(e_0)$ is empty for all i ($1 \leq i \leq n$), then e_0 is defined to be the uppermost node. ■

Now, denote $t_q^{e_0}$ the subtree of t containing all of the touched edges. From the lemma straightforwardly follows that $t_q^{e_0}$ is rooted. Now, for simulating XPath$_0$ expressions we shall use special structural recursions constructing only ψ edges.

Definition 2 *Let* $f = f_1, \ldots, f_n$ *be a structural recursion constructing only* ψ *edges and* t *a document tree. For an edge* $(u, a, v) \in E.t$ *we say that* f stops *on* (u, a, v), *if there is an edge* $(f_i, p, \varphi) \in E.U_f$, *s.t.* $((f_i, u), p \wedge a, (\varphi, v)) \in G_{f,t}$, *and as a result of this edge a* ψ *edge is constructed* ($\varphi \in \{f_1, \ldots, f_n, w\}$).

Definition 3 *Let* q *be an XPath*$_0$ *expression and* f *a structural recursion. Then* f *is* equivalent with q, *in notation* $f \simeq q$, *if for all document tree* t *and* $e_0, e \in E.t$, $e \in q(e_0)$ *iff* f *called on* $t_q^{e_0}$ *stops on* e.

Note that the definition still makes sense, when $q(e_0)$ is empty.

XPath$_0$ without predicates

In this subsection, when we talk about an XPath$_0$ expression q, we always assume that it is without predicates. Consider the following expression

$$q_2 := \texttt{self::a/child::b/desc::d/anc::c},$$

where `anc` is an abbreviation of `ancestor`. Here we do not know whether a or b is an ancestor or a descendant of c. Hence when we are to simulate q_2, we have to construct structural recursions checking only whether an a edge has a b child, having a descendant d edge etc.

Furthermore, structural recursions run in a top-down manner, thus if a structural recursion simulating q_2 stops on appropriate c edges, it should have already checked whether there are d edges below these c edges, not to mention other relations among other edges.

Finally, we should keep in mind that if for $x, y, z \in E.t$, $lab(y) = lab(z)$, $x\chi y, x\chi z$ both holds, where χ, again, denotes an axis, then the corresponding simulation should both return y and z. That is, in the document

```
<a><b><b>...</b></b></a>
```

there are two descendant b elements, and both should be returned in a simulation of `self::a/desc::b`.

From now on, we assume that axis `self` only appears as the first axis of an expression. Since subexpression `a/self::b` is unsatisfiable, while in subexpressions `a/self::a`, `a/self::*`, `self::a`, `self::*` can be omitted, our assumption is justified.

XPgraphs. Let q be an $XPath_0$ expression. An edge test a precedes edge test b in q, if a is written first in left-to-right order. The last edge test is called *aim*. If there are more than one a edge test in q, then the first occurrence is indexed a_1, the second a_2 etc. Thus, in the rest of this section we shall assume that all edge tests are different.

For representing the relations between edge tests we construct an auxiliary graph, *XPgraph*. We denote the XPgraph of q with XP_q. The nodes of XP_q are labelled with edge tests of q. There is an edge from a to b, if either `a/desc::b`, or `b/anc::a` is a subexpression of q. In case of `a/child::b`, `b/par::a` this edge is labelled with ch, par respectively. Clearly, XP_q is without cycles. The representation of the aim is called *aim node*. In XP_q node a precedes node b, if edge test a precedes edge test b. A node a in XP_q is an *upper node*, if it has no ingoing edges.

Example 6 *For* $q_3 =$ `self::a/child::b/desc::d/par::a`, XP_{q_3} *is given in Fig. 10(a).*

XPgraphs with one upper node. For the simulation of q we use XP_q. First we suppose that it has only one upper node. First we illustrate the method with an example.

Example 7 *The representation of* $q_4 :=$ `self::a/anc::b/child::d/child::c`, $f_{q_4} = (f_a, f_b, f_c, f_d)$, *is the following (*$XP_{q_4}$ *can be found in Fig. 10(b)):*

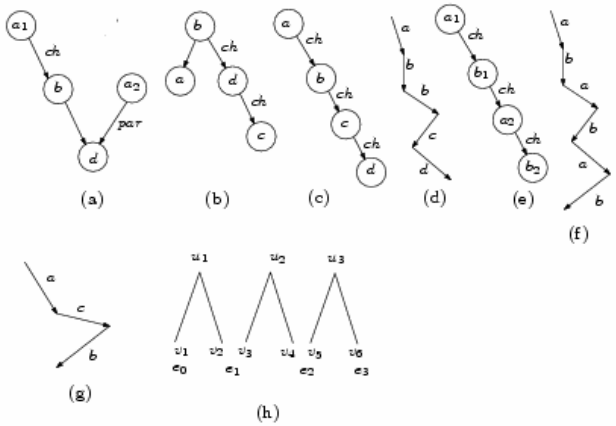

Figure 10: (a) XP_{q_3}. (b) XP_{q_4}. (c) The XPgraph of Example 8 (d) The document tree of Example 8. (e) The XPgraph of Example 9. (f) The document tree of Example 9. (g) A document tree.

$$\mathsf{f_b}\colon (\{\mathsf{b}:\mathsf{t}\}) = \textit{if } \textit{n.i.}(\mathsf{f_a}(\mathsf{t})) \textit{ then } \mathsf{f_d}(\mathsf{t}) \textit{ else } \mathsf{f_b}(\mathsf{t})$$
$$(\{*:\mathsf{t}\}) = \mathsf{f_b}(\mathsf{t})$$

$$\mathsf{f_c}\colon (\{\mathsf{c}:\mathsf{t}\}) = \{\psi:\{\}\}$$
$$(\{*:\mathsf{t}\}) = \mathsf{f_b}(\mathsf{t})$$

$$\mathsf{f_a}\colon (\{\mathsf{a}:\mathsf{t}\}) = \{\psi:\{\}\}$$
$$(\{*:\mathsf{t}\}) = \mathsf{f_a}(\mathsf{t})$$

$$\mathsf{f_d}\colon (\{\mathsf{d}:\mathsf{t}\}) = \mathsf{f_c}(\mathsf{t})$$
$$(\{*:\mathsf{t}\}) = \mathsf{f_b}(\mathsf{t})$$

In the simulation to each $\mathsf{a} \in \mathsf{V.XP_q}$ we assign a structural function $\mathsf{f_a}$. Each such function contains two rows: the default case and another row corresponding to its label. With the latter we specify what should happen, when the "desired label" is reached. For instance, the function representing the aim node, in accordance with the definition of the equivalence of XPath_0 expressions and structural recursions, should construct a ψ edge. In the default case, we can control whether we are to simulate a `desc`, `anc` axis, or a `child`, `par`. In our example, $\mathsf{f_c}$ calls $\mathsf{f_b}$ in the default case, since the b edge should be followed by a c edge immediately. For similar reasons, $\mathsf{f_d}$ also calls $\mathsf{f_b}$ in its default case. On the other hand, $\mathsf{f_a}$ should call itself, since there may be arbitrary number of edges between b and a.

The upper node of XP_{q_4} is b, hence the process of an input should start

with f_b, thus it is defined to be the upper structural function. With the n.i.$(f_a(t))$ condition we check whether the examined b edges have an a descendant. Note that in functions representing a leaf different from the aim node, in the non-default row we should also construct a ψ edge, since the check of a n.i. condition ends there.

Formally, for edges $(a, b) \in E.XP_q$ in the default case f_a calls itself. For edges $(a, par, b) \in E.XP_q$, in the default case the process should stop with $\{\}$. On the other hand, in case of $(b, ch, a) \in E.XP_q$, first we have to find the maximal child-chain, $u_1, \ldots, u_m$ s.t. $u_i \in V.XP_q, (u_i, ch, u_{i+1}) \in E.XP_q, u_m = a, (u_{m-1} = b)$. Then, in the default case f_a should call f_{u_1}.

In the a row $(\{a : t\}) = \ldots)$, (i) if a is a leaf, or the aim node, a ψ edge should be constructed. If there is a condition (see below), then this ψ edge should be constructed in the then-branch.

(ii) If (a, b) or (a, ch, b) or (a, par, b) is the only outgoing edge of a, then, if a is the aim node, then n.i.$(f_b(t))$ is called, and a ψ edge should be constructed in the then-branch. In the else branch f_a should call itself. In case of edge (a, par, b) and for edge (a, b), if a is the ancestor of b, a is surely the aim node. Otherwise, if a is not the aim node, f_b should be called. Again, in case of edge (a, ch, b) and for edge (a, b), if b is the descendant of a, a is surely not the aim node.

(iii) If a has another outgoing edge beside $(a, b), (a, par, b)$ or (a, ch, b), then if the aim node is not reachable through b on a directed path, then f_b should be called in a n.i. condition. Otherwise, it should be called in the then-branch. In the else-branch, again, f_a should call itself. Note that in this case a is an upper node.

Denote $\tilde{f}_q$ the result structural recursion. Here, the only upper structural function is the structural function of the upper node. In order to describe the connection between q and $\tilde{f}_q$, we introduce two notions.

Definition 4 *For an XPath$_0$ expression* $q = \chi_1 :: \tau_1 / \ldots / \chi_n :: \tau_n$*, document tree* t*,* $e_0 \in E.t$*,* $e \in E.t$ *is an* uppermost result element of $q_{\tau_i}(e_0)$*, if (i)* $e \in q_{\tau_i}(e_0)$*,* e *is in a result chain, (ii) there does not exist any* $e' \in E.t$ *s.t.* e' *is an ancestor of* e*,* $e' \in q_{\tau_i}(e_0)$*, and* e' *is also in a result chain.*

Definition 5 *Let* q *be an XPath$_0$ expression without predicates,* t *a document tree,* $(u, a, v) = e \in E.t$*. Then, we say that* the a row of f_a in $\tilde{f}_q$ is called successfully on e*, if (i)* $((f_a, u), a, (\theta, v)) \in E.U_f \sqcap t$*, (ii) if there is a condition in the* a *row, it is satisfied* $(\theta \in \{f_\sigma, w\}, \sigma \in \Sigma)$*.*

With $e \in R^t(a, f_a)$, or shortly $e \in R(a, f_a)$, if the input is clear from the

context, we denote that the a row of f_a is called successfully on e.

Lemma 2 *Let* $q = \chi_1 :: \tau_1/\ldots/\chi_n :: \tau_n$ *be an XPath*$_0$ *expression s.t.* XP_q *has only one upper node. Let* t *be a document tree,* $e_0 \in E.t$*. Consider* τ_j*, and* $e \in E.t$ *s.t.* $lab(e) = \tau_j$*, and* e *does not have any ancestor* e' *s.t.* e' *is an uppermost result element of* $q_{\tau_j}(e_0)$*. Then* $e \in q_{\tau_j}(e_0)$ *iff* $e \in R(\tau_j, f_{\tau_j})$ (f_{τ_j} *is in* $\tilde{f}_q, 1 \leq j \leq n$)*.*

Proof. Suppose that $(u, \tau_i, v) = e^1$ is the uppermost element of $t_q^{e_0}$. From this it follows that τ_i is the upper node of XP_q, and $((f_{\tau_i}, u), \tau_i, (\theta, v)) \in E.U_f \sqcap t$ ($\theta \in \{f_\sigma, w\}, \sigma \in \Sigma$).

(a) If $e^1 \in q_{\tau_i}(e_0)$, then there exists a sequence of edges $e_0, \ldots, e_i$ s.t. $e_i = e^1$, $e_j \chi_{j+1}, e_{j+1}$ ($0 \leq j \leq i-1$). Since τ_i is the upper node, f_{τ_i} calls $f_{\tau_{i-1}}$ on e in a n.i. condition. Now, suppose that χ_{i-1} is `anc`. Then the τ_{i-1} row of $f_{\tau_{i-1}}$ is only instantiated when the first edge under e^1 with label τ_{i-1} is reached. If τ_{i-1} is a leaf in XP_q, a ψ edge is constructed, and the n.i.condition is satisfied, consequently, e^1 is in $R(\tau_i, f_{\tau_i})$.

If χ_{i-1} is `par`, the reasoning is similar. The proof can be continued inductively on the rest of the edge tests before τ_i. At the end we get that $e^1 \in R(\tau_i, f_{\tau_i})$

For the other direction, if $e^1 \in R(\tau_i, f_{\tau_i})$, and there is no condition in the τ_i row, then τ_i is not preceded by any edge test in q, hence $i = 1$. Obviously, $e^1 \in q_{\tau_i}(e_0)$. Otherwise, the condition is satisfied. Again, if τ_{i-1}`/anc::`τ_i (τ_{i-1}`/par::`τ_i) is a subexpression of q, then a trivial analysis of $f_{\tau_{i-1}}$ shows that it is guaranteed that e^1 has a descendant (child) with label τ_{i-1}. The proof can be continued inductively. At the end we get that $e^1 \in q_{\tau_i}(e_0)$. Hence, the statement of the lemma holds for e^1.

Beside the supposition that $e^1 \in q_{\tau_i}(e_0)$, assume further that τ_{i-1}`/anc::`τ_i is a subexpression of q. Let e^2 be a descendant of e^1 s.t. $lab(e^2) = \tau_{i-1}$, and there is not any ancestor e^3 of e^2 s.t. e^3 is a descendant of e^1, $lab(e^3) = \tau_{i-1}$. Such edges will be called *first* τ_{i-1} *descendants*. The previous reasoning shows that the τ_{i-1} row of $f_{\tau_{i-1}}$ is called on e^2. Furthermore, from the rewriting rules we also know that this τ_{i-1} row does not contain any condition. Consequently, $e^2 \in R(\tau_{i-1}, f_{\tau_{i-1}})$. Additionally, since $e^1 \in q_{\tau_i}(e_0)$, $e^2 \in q_{\tau_{i-1}}(e_0)$. Hence, the statement also holds for e^2. The proof is similar, when τ_{i-1}`/par::`τ_i is a subexpression of q. Thus, it can be shown inductively that the lemma holds for all edge tests before τ_i in q.

If τ_i`/desc::`τ_{i+1} is a subexpression, then f_{τ_i} calls $f_{\tau_{i+1}}$, which again finds the first τ_{i+1} descendants of e^1. Again in its τ_{i+1} row $f_{\tau_{i+1}}$, even if τ_{i+1} is the

aim node, does not have any condition. Consequently, the lemma also holds for these edges.

In case of $\tau_i/\texttt{child::}\tau_{i+1}$, if e^2 is a child of e^1 with label τ_{i+1}, then the statement is true for e^2. Otherwise, $f_{\tau_{i+1}}$ calls f_{τ_i} in its default case, and the whole previous reasoning can be applied to the first τ_i descendants of e^2.

(b) If $e^1 \notin q_{\tau_i}(e_0)$, then the condition of the τ_i row of f_{τ_i} is not satisfied, hence f_{τ_i} is called again in the else branch. Again, the reasoning is similar for the first τ_i descendants.

Furthermore, if edge e is an uppermost result element of $q_{\tau_j}(e_0)$, then it is easy to show, that f_{τ_j} is not instantiated on any descendants of e. ■

In other words the lemma says that $\tilde{f}_q$, for an arbitrary input tree and a fixed edge e_0, finds all of those "uppermost" edges that may be touched, or may be elements of a result chain of q called on e_0.

Corollary 1 *Keeping the assumptions of Lemma 2, for document tree* t*, edge test* τ_j*,* $e \in E.t$*, where* $\texttt{lab}(e) = \tau_j$*, and for an arbitrary edge* $\bar{e}$ *of* t*, if* e *does not have any ancestor* e' *s.t.* e' *is an uppermost result element of* $q_{\tau_j}(\bar{e})$*. Then* $e \in q_{\tau_j}(\bar{e})$ *iff* $e \in R(\tau_j, f_{\tau_j})$ (f_{τ_j} *is in* $\tilde{f}_q, 1 \le j \le n$)*.*

Note that the only difference between Lemma 2 and Corollary 1 is that in the latter case we do not fix edge e_0, where the evaluation of q should start.

A document tree t is called *simple*, if for an arbitrary edge e, where $\texttt{lab}(e) = \sigma$, e does not have any σ ancestor in t.

Corollary 2 *Keeping the above assumptions and notations, if* t *is simple, then* q *is equivalent with* $\tilde{f}_q$*.*

Corollary 2 shows that $\tilde{f}_q$ simulates q correctly for a large and practically the most important class of document trees.

To establish the equivalence for all document trees, we have to develop a modified version of function f_σ to be able to process σ edges that are in a descendant-ancestor relationship. For this we introduce a method called the *repetition of* σ. Informally, in the σ row we call f_σ or the appropriate structural function again. As an example of repetition, consider $f = (f_{a_1}, f_b, f^1_d)$ of Example 3 that simulates `self::a/child::b/desc::d`. There, we have repeated d in f^1_d.

Formally, in the repetition of τ_i, if neither the ingoing, nor the outgoing edge of τ_i is labelled with $\texttt{ch}$ or $\texttt{par}$, then in its τ_i row f_{τ_i} should also call itself. This means that on the right side (in the then branch, if it exists) instead of $\{\psi : \{\}\}$ or $f_{\tau_{i+1}}(t)$, $\{\psi : \{\}\}@f_{\tau_i}(t)$ or $f_{\tau_{i+1}}(t)@f_{\tau_i}(t)$ should be written.

If τ_i is in a path $pa = u_1 \dots u_m$ in XP_q s.t. pa has either solely ch, or solely par edges, then (suppose that pa is maximal in terms subsumption with respect to this property), then in the repetition of $\tau_i\ f_{u_1}$ should call itself in its u_1 row.

In order to describe why we have chosen this construction we show two examples.

Example 8 *First, consider the XPgraph of Fig. 10(c), and suppose that we are to repeat* b*, but here* f_b *calls itself in its* b *row. (According to our definition* f_a *should call itself in its* a *row.)*

$$\begin{aligned} f_a\colon (\{a:t\}) &= f_b(t) \\ (\{*:t\}) &= f_a(t) \end{aligned} \qquad \begin{aligned} f_b\colon (\{b:t\}) &= f_c(t)@f_b(t) \\ (\{*:t\}) &= f_a(t) \end{aligned}$$

$$\begin{aligned} f_c\colon (\{c:t\}) &= f_d(t) \\ (\{*:t\}) &= f_a(t) \end{aligned} \qquad \begin{aligned} f_d\colon (\{d:t\}) &= \{\psi:\{\}\} \\ (\{*:t\}) &= f_a(t) \end{aligned}$$

It is not difficult to see that for the document tree of Fig. 10(d) this structural recursion constructs a ψ-edge, though it should not. Here the problem is that it is not checked whether the second b edge has an a parent. This shows that in the repetition the whole check should start from the beginning, hence f_{u_1} should be called.

Example 9 *Secondly, for the XPgraph of Fig. 10(e) we repeat* a_2*, but* f_{a_1} *is called in the* a *row of* f_{a_2} *instead of the* a *row of* f_{a_1}*.*

$$\begin{aligned} f_{a_1}\colon (\{a:t\}) &= f_{b_1}(t) \\ (\{*:t\}) &= f_{a_1}(t) \end{aligned} \qquad \begin{aligned} f_{b_1}\colon (\{b:t\}) &= f_{a_2}(t) \\ (\{*:t\}) &= f_{a_1}(t) \end{aligned}$$

$$\begin{aligned} f_{a_2}\colon (\{a:t\}) &= f_{b_2}(t)@f_{a_1}(t) \\ (\{*:t\}) &= f_{a_1}(t) \end{aligned} \qquad \begin{aligned} f_{b_2}\colon (\{b:t\}) &= \{\psi:\{\}\} \\ (\{*:t\}) &= f_{a_1}(t) \end{aligned}$$

Again, for the document tree of Fig. 10(f) only one ψ edge is constructed as a result of the second b edge, though as a result of the third b edge another ψ edge should also be constructed. This is because f_{a_1} is not called on the second a edge. This shows that in the repetition f_{u_1} should be called in its own u_1 row.

Suppose now that in Example 9 we call f_{a_1} in its a row. Then for the document tree of Fig. 10(g) after the c edge two "instances" of f_{a_1} are called on b, although only one should process this edge. Hence, we should slightly change the rewriting rules, when a node of a child-chain is to be repeated. In this case, in the default cases $f_{u_2}, \ldots, f_{u_m}$ instead of calling f_{u_1} the empty graph should be constructed. With this the XPgraph of Fig. 10(e) should be rewritten as follows.

$$f_{a_1} : (\{a : t\}) = f_{b_1}(t)@f_{a_1}(t) \qquad f_{b_1} : (\{b : t\}) = f_{a_2}(t)$$
$$(\{* : t\}) = f_{a_1}(t) \qquad (\{* : t\}) = \{\}$$

$$f_{a_2} : (\{a : t\}) = f_{b_2}(t) \qquad f_{b_2} : (\{b : t\}) = \{\psi : \{\}\}$$
$$(\{* : t\}) = \{\} \qquad (\{* : t\}) = \{\}$$

Finally, if we are to repeat $\tau_i = u_k$ and $\tau_j = u_r$, $(1 \leq k, l \leq m)$, then it is enough to call f_{u_1} in its u_1 row once.

Lemma 3 *For a given XPath$_0$ expression $q = \chi_1 :: \tau_1/\ldots/\chi_n :: \tau_n$, where XP_q has only one upper node, document tree t and $e_0 \in E.t$, $e \in E.t$ is in $q_{\tau_i}(e_0)$ iff $e \in R(\tau_i, f_{\tau_i})$, where we have repeated τ_i in $\tilde{f}_q$.*

Proof. The statement straightforwardly follows from the proof of Lemma 2 and from the previous consideration. ■

Now, repeat τ_n in $\tilde{f}_q$. We denote the result with f_q.

Corollary 3 *Keeping the above notations, $q \simeq f_q$.*

Note that from Lemma 2 and Lemma 3 it also turns out that, when we simulate q with f_q, and traverse t top-down, in most cases we only look for the first elements that are in a result chain. When we find one, the corresponding part of the process stops. It seems hard to find such an implementation strategy that would not process these elements in a top-down traverse. Thus, we can say that we only process those edges that are necessary to process.

XPgraphs with several upper nodes. Unfortunately, till now, we have not found any rewriting technique with which we could simulate an XPath$_0$ expression, whose XPgraph has several upper nodes, with a structural recursion not using registers. However, with registers, the simulation is almost straightforward.

Let $u_1, \dots u_m$ be the upper nodes of XP_q. For u_i denote $XP_q^{u_i}$ the subtree reachable from u_i in XP_q $(1 \leq i \leq m)$. Clearly, each $XP_q^{u_i}$ represents a "subquery" of q. For

$$q_3 = \texttt{self::a/child::b/desc::d/par::a}$$

these subqueries are `self::a/child::b/desc::d` and `desc::d/par::a` (cf. XP_{q_3} in Fig. 10(a)). With the results of the previous paragraphs, the corresponding structural recursion, $f_q^{u_i}$, can be given. Note that a leaf a of XP_q may appear in both $XP_q^{u_i}$ and $XP_q^{u_{i+1}}$ $(1 \leq i \leq m-1)$. To differentiate between the appropriate structural functions, we denote them f_a^i, f_a^{i+1}, respectively. With this for such leaves we require $X^a_{f_a^i} = X^a_{f_a^{i+1}}$. The intuition is clear, simply, we are to connect structural recursions $f_q^{u_i}$ and $f_q^{u_{i+1}}$.

However, here, in the structural recursion we have to repeat the preceding a leaves as well, otherwise it is not difficult to see, we may lose elements of the result. We denote these new structural recursions $f_q^{u_i}$ and $f_q^{u_{i+1}}$ again. f_q is constituted by the structural functions of $f_q^{u_i}$-s $(1 \leq i \leq n)$. $f_{u_1}, \dots, f_{u_m}$ are the upper structural functions and, since the aim node is reachable from u_m, we designate f_{u_m} to be the root structural function of f_q. It is important to note that the size of f_q is linear in the size of q.

As an example, consider $f_{q_3} = (f_{a_1}, f_b, f_d^1, f_{a_2}, f_d^2)$ of Example 3 with register restriction $X^d_{f_d^1} = X^d_{f_d^2}$. Here, the root structural function is f_{a_2}.

Theorem 1 *Let q be an XPath$_0$ expression without predicates. Then $f_q \simeq q$.*

Proof. For $e \in E.t$ with $e \in R^t_{reg}(a, f_a)$, or shortly $e \in R_{reg}(a, f_a)$, we are to indicate that (i) $e \in R^t(a, f_a)$ (ii) e satisfies all register restrictions $X^a_{f_a} = X^a_{f_\sigma}$ $\sigma \in \Sigma$. (For f_a^i, register restrictions $X^a_{f_a^i} = X^a_{f_a^{i+1}}$ should also be satisfied.)

Assume now that $q = \chi_1 :: \tau_1/\dots/\chi_n :: \tau_n$, XP_q has two upper nodes, u_1, u_2, and the common leaf represents τ_i $(1 \leq i \leq n-1)$. $q' := \chi_{i+1} :: \tau_{i+1}/\dots/\chi_n :: \tau_n$. Clearly, for an arbitrary document tree t, $e_0 \in E.t$, $e \in q(e_0)$ iff there exists $\bar{e} \in E.t$, $\bar{e} \in q_{\tau_i}(e_0)$ s.t. $e \in q'(\bar{e})$. Hence, using the results of Corollary 1 and Lemma 3 $e \in q(e_0)$ iff $e \in R^t_{reg}(\tau_n, f_{\tau_n})$. Since the root structural function is f_{u_2}, f_q may only stop on edges in $R^t_{reg}(\tau_n, f_{\tau_n})$. (If there was not any root structural function, since τ_i is the aim node of $XP_q^{u_1}$, f_q would also stop on edges in $R^t_{reg}(\tau_i, f^1_{\tau_i})$, thus q and f_q would not be equivalent (see Definition 3)). All in all, we get that $e \in q(e_0)$ iff f_q stops on e. The proof can be continued inductively on the number of upper nodes of XP_q in the same way. ■

Now, one may find too costly that for example in f_{q_3} we traverse the whole input with (f_{a_2}, f^1_d), with which, remember, we seek for a edges with d children. There may be too many of such a edges that are not included in the result, and another implementation would not take care of them, since in the previous steps it has already found the appropriate d edges with a b ancestor, whose parent is a. On the reverse side of the coin, in most cases elements do not have descendants with the same name, in these scenarios it is enough to check the first a elements. This information can be obtained when the structure of the input is given by an XML Schema or a DTD. In [4] we show how DTDs and extended DTDs work together with our methods and handle, among other things, the preceding problem.

XPath$_0$ with predicates

In order to ease the notation, we shall assume that

$$q = \chi_1 :: \tau_1[p_1]/\ldots/\chi_n :: \tau_n[p_n].$$

Here, τ_i is called the *base* of pr_i-s ($1 \le i \le n$), while

$$q^{sk} = \chi_1 :: \tau_1/\ldots/\chi_n :: \tau_n$$

is called the *skeleton (of* q*)*. First, we suppose that each p_i consists of a single atomic condition, $q^{p_i} = c$ or $q^{p_i} = x$. Furthermore, for a moment, we also assume that all q^{p_i}-s are of the form `self::*` or `self::`τ_i.

Then, when we construct f_q, the τ_i row of f_{τ_i} should be completed with $val(\tau_i) = c$ or $val(\tau_i) = x$ condition in accordance with the conditions of p_i-s. (If there is already a condition, then the conjunction of the conditions should be taken.) In the else branch (if it has not given previously) f_{τ_i} should call itself.

Lemma 4 *Keeping the previous notation* $q \simeq f_q$.

Proof. The statement trivially follows from Theorem 1 and of the additional rule of construction. ■

XPath$_0$ without embedded predicates. Loosening our restrictions, we only assume now that the predicates consist of a single atomic condition, whose XPath$_0$ expression does not contain predicates. Using the indexing of edge tests of the previous subsection, again, we shall assume that the edge tests of q are all different from each other. Straightforwardly, the XPath$_0$

expression of a predicate $q^p = \chi_1 :: \tau_1/\ldots/\chi_k :: \tau_k$ can be rewritten as $\texttt{self::}\tau_0/\chi_1 :: \tau_1/\ldots/\chi_k :: \tau_k$, where τ_0 is the base of q^p. Similarly, condition $q^p = \theta$ means the same as

$$\texttt{self::}\tau_0/\chi_1 :: \tau_1/\ldots/\chi_k :: \tau_k\texttt{[self::*=}\theta\texttt{]}\ (\theta \in \{c, x\}),$$

consequently q^p can be simulated using the technique of the previous paragraph. In what follows, we shall always use this rewriting.

Next, by means of register restrictions, we are to "join" the structural recursions of predicates with the structural recursion of the skeleton using bases as connection points. Hence, when we rewrite q^p, we should take the base as the aim node, which was the starting point of q^p originally. This means that we move in the opposite direction. Consequently, if in q^p χ_j is `ch` or `par`, then in the XPgraph of q^p we should change the `ch` labels to `par` labels and vice versa ($1 \leq j \leq k$). With this $f_{q^{p_i}}$, or shortly f_{p_i}, can be constructed. The structural function corresponding for the predicate of base τ_i will be denoted $f^{p_i}_{\tau_i}$.

In the structural recursion of the skeleton, f_{sk}, we repeat each base (remember that in a child- or parent-chain $u_1 \ldots u_m$, if we are to repeat u_k, u_s, then f_{u_1} is called in its u_1 row only once ($1 \leq k, s \leq m$)). Again, if we did not do so, we may lose elements of the result.

To use bases as connection points, we add the restrictions $X^{\tau_i}_{f_{\tau_i}} = X^{\tau_i}_{f^{p_i}_{\tau_i}}$. f_q is constituted by the structural functions of f_{p_i}-s, f_{sk} and the preceding register restrictions. Its upper structural functions are the structural functions of upper nodes. Moreover, suppose that XP^u is the XPgraph containing the aim node of the skeleton. Then f_u is designated to be the root structural function. As an example consider the rewriting of q_1, $f = (f_{a_1}, f_b, f^1_d, f_{a_2}, f^2_d, f^{pr_1}_c, f^{pr_1}_{a_1})$ of Example 3.

Theorem 2 *Let* q *be an XPath*$_0$ *expression s.t. each edge test has at most one predicate, and the XPath*$_0$ *expressions of these predicates do not contain predicates, then* $q \simeq f_q$.

Proof. The theorem straightforwardly follows from the previous considerations. ■

Corollary 4 *Let* q *be an XPath*$_0$ *expression without variables. Then for an arbitrary document tree* t, $f_q(t)$ *can be constructed in* $O(|f||t|)$ *time. If the size of variables, i.e., the size of the list of edges which the variable is equal to, is restricted with a constant, the evaluation of* $f_q(t)$ *still works in linear time. In worst case scenarios the construction can be accomplished in* $O(|f||t|^2)$ *time.*

General case. Firstly, we allow Boolean combinations of atomic conditions in the predicates. Without loss of generality, we may suppose that base τ_r has predicate $(\neg)(q^{p_1} = b_1)\theta_1 \dots \theta_k(\neg)(q^{p_k} = b_k)$; with $(\neg)$ we indicate that there may be a negation ($\theta_i \in \{\wedge, \vee\}, 1 \leq i \leq k$). Then the corresponding register restriction is of the form: $\hat{R}eg_1\theta_1 \dots \theta_k\hat{R}eg_k$, where $\hat{R}eg_i \in \{Reg_i, \neg Reg_i\}$, and Reg_i is $X^{\tau_r}_{f_{\tau_r}} = X^{\tau_r}_{f^{p_{r_i}}_{\tau_r}}$. In $\hat{R}eg_i\ Reg_i$ is negated if the corresponding $q^{p_i} = b_i$ condition is negated.

Secondly, if we allow predicates in the $XPath_0$ expressions of predicates, then the rewriting algorithm can be continued recursively. Again, the size of the result structural recursion is linear in the size of the simulated $XPath_0$ expression.

5 Rewriting of XSLT$_0$

As in the previous section, first we have to define the equivalence of an $XSLT_0$ program and a structural recursion.

Definition 6 *Let* t, t' *be document trees. Then* t *is equivalent with* t'*, if there is a one-to-one mapping* $\phi : V.t \to V.t'$ *s.t. (i)* $\phi(u_0) = u'_0$*, where* u_0, u'_0 *denote the roots of* t *and* t'*, respectively. (ii) For* $e = (u, a, v) \in E.t$*,* $e' = (\phi(u), a, \phi(v))$ *is also in* $E.t'$*, and* $val(e) = val(e')$*. (iii) If for* $e^1 = (u^1, a^1, v^1), e^2 = (u^2, a^2, v^2) \in E.t$*,* e^1 *precedes* e^2*, then* $(\phi(u^1), a^1, \phi(v^1))$ *also precedes* $(\phi(u^2), a^2, \phi(v^2))$*.*

Definition 7 *For an arbitrary XSLT$_0$ program* P *and structural recursion* f*,* P *is equivalent with* f*, if for all document tree* t*,* $\tau_P(t)$ *is equivalent with* $f(t)$*.*

Rewriting of (m, σ)-rules

First, we are to simulate a single (m, σ)-rule, T, with a given variable assignment. For this we assume that T does not construct anything just selects a list of edges for further processing in mode r. An instantiated template then simply constructs a ψ edge and stops. Furthermore, remember that each program contains a special $(st, /)$-rule, which is called on the document edge and constructs a `result` edge as the document edge of the output. Here, we also suppose that this rule invokes the next template in σ descendants in mode m, i.e., T is called. The corresponding program is denoted P_T. With these suppositions we get that P_T constructs a star of ψ edges.

Example 10 *Consider this rather artificial example, where* T *is called on* $\mathtt{a}$ *edges, and if the edge in question has a* $\mathtt{c}$ *ancestor with value* 5*, the next template is invoked on parent* $\mathtt{b}$*, otherwise on children* $\mathtt{c}$*.*

template $\mathsf{st}(/, \epsilon)$ $\quad (\mathsf{T}_{st})$
  return
    if true *then* $\{\mathsf{result} : \{\}\}$ *at-expr:* $\mathfrak{m}(\mathtt{desc::a}, \epsilon, \epsilon)$;
end.

template $\mathfrak{m}(\mathtt{a}, \epsilon)$ $\quad (\mathsf{T})$
  return
    if $\mathtt{anc:c} = 5$ *then* $\{\}$ *at-expr:* $\mathsf{r}(\mathtt{par::b}, \epsilon, \epsilon)$;
    if true *then* $\{\}$ *at-expr:* $\mathsf{r}(\mathtt{child::c}, \epsilon, \epsilon)$;
end.

template $\mathsf{r}(\mathtt{b}, \epsilon)$ $\quad (\mathsf{T}_1)$
  return
    if true *then* $\{\psi : \{\}\}$
end.

template $\mathsf{r}(\mathtt{c}, \epsilon)$ $\quad (\mathsf{T}_2)$
  return
    if true *then* $\{\psi : \{\}\}$
end.

Remember that an $(\mathfrak{m}, \sigma)$-rule contains conditions of the form: if $\mathtt{c}_i$ then z_i; where $z_i \in \mathcal{F}^{\Delta}(\mathcal{AT})$ (forests with edge labels from Δ and with possible at-expression leaf labels) $(1 \leq i \leq k)$. Since T constructs nothing, each z_i is a node with at-expression $\mathsf{r}(\mathtt{p}, \epsilon, \epsilon)$, where, remember, $\mathtt{p}$ is an XPath_0 expression, the first ϵ means T has no parameters, the second ϵ indicates that there is not any variable assignment. In the sequel $\mathtt{p}$ will be referred as $\mathtt{xp}(z_i)$.

It is easy to see now that such a condition works in the same way as $\mathtt{q}_i :=$ $\mathtt{self::}\sigma\mathtt{[}c_i\mathtt{]/xp}(z_i)$. Denote XP_i the XPgraph of $\mathtt{q}_i$. Then we may speak of the aim node of the i^{th} condition. Clearly, the corresponding structural recursion of $\mathtt{q}_i$, ${}_{\mathsf{T}}\mathtt{f}_{q_i}$ can be constructed (here T in the subindex implies that the structural recursions simulates a condition in template T). Remember that ${}_{\mathsf{T}}\mathtt{f}_{q_i}$ is divided into two structural recursions representing

$$\mathtt{self::}\sigma/c_i \text{ (predicate) and } \mathtt{self::}\sigma/\mathtt{xp}(z_i) \text{ (skeleton).}$$

Denote them ${}_{\mathsf{T}}\mathtt{f}_{c_i}$ and ${}_{\mathsf{T}}\mathtt{f}_{z_i}$, respectively.

Owing to its distinguished role, the representation of σ will be called *matching node*. Since we are to call structural recursion ${}_{\mathsf{T}}\mathtt{f}_{q_i}$ on every possible subtree $\{\sigma : t\}$ of the input, σ should also be repeated. If $\mathtt{c}_i$ is a real condition,

then σ is also a base, and according to the rewriting rules of XPath_0 expression with predicates, every base should be repeated. However, if c_i is constant $\mathtt{true}$, then we should repeat σ explicitly. In the rewriting of the program of Example 10, in order to be able to simulate P_T, we introduce two new constructions.

$$
\begin{array}{l}
{}_{T_{st}}f_{/}:(\{/:t\}) = \{\mathtt{result}:\{{}^{z_1}_{T}f_b(t),{}^{z_2}_{T}f_a(t)\}\} \\
\\
{}^{c_1}_{T}f_c:\ (\{c:t\}) = \text{if } val(c)=5 \text{ then } {}^{c_1}_{T}f_a(t) \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \text{else } {}^{c_1}_{T}f_c(t) \\
\qquad\quad (\{*:t\}) = {}^{c_1}_{T}f_c(t) \\
\\
{}^{c_1}_{T}f_a:\ (\{a:t\}) = \{\psi:\{\}\}@{}^{c_1}_{T}f_a(t) \\
\qquad\quad (\{*:t\}) = {}^{c_1}_{T}f_a(t) \\
\\
{}^{z_1}_{T}f_b:\ (\{b:t\}) = \text{if n.i.}({}^{z_1}_{T}f_a)(t)) \text{ then } \{\psi:\{\}\}@{}^{z_1}_{T}f_b(t) \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \text{else } {}^{z_1}_{T}f_b(t) \\
\qquad\quad (\{*:t\}) = {}^{z_1}_{T}f_b(t) \\
\\
{}^{z_1}_{T}f_a:\ (\{a:t\}) = \{\psi:\{\}\} \\
\qquad\quad (\{*:t\}) = \{\},\qquad X^{a}_{{}^{c_1}_{T}f_a} = X^{a}_{{}^{z_1}_{T}f_a} \\
\\
{}^{z_2}_{T}f_a:\ (\{a:t\}) = {}^{z_2}_{T}f_c(t)@{}^{z_2}_{T}f_a(t) \qquad {}^{z_2}_{T}f_c:\ (\{c:t\}) = \{\psi:\{\}\} \\
\qquad\quad (\{*:t\}) = {}^{z_2}_{T}f_a(t) \qquad\qquad\qquad\qquad\quad (\{*:t\}) = \{\}
\end{array}
$$

if $e \in \tilde{X}^{a}_{{}^{c_1}_{T}f_a}$ then delete ${}^{z_2}_{T}f_a(e)$
else if $\mathtt{true}$ then delete ${}^{z_1}_{T}f_a(e)$

The first of these new constructions is in the right side of the / row of ${}_{T_{st}}f_{/}$. Namely, the leaf of $\{\mathtt{result} : \{\}\}$ is labelled with a set of structural functions ${}^{z_1}_{T}f_b(t),{}^{z_2}_{T}f_a(t)$ instead of a single structural function (cf. Fig. 6), in notation $\{\mathtt{result} : \{{}^{z_1}_{T}f_b(t),{}^{z_2}_{T}f_a(t)\}\}$. With this we indicate that both the results of ${}^{z_1}_{T}f_b,{}^{z_2}_{T}f_a$ called on t should be appended to the $\mathtt{result}$ edge. Here, ${}^{z_1}_{T}f_b,{}^{z_2}_{T}f_a(t)$ are the root structural functions of ${}_{T}f_{z_1},{}_{T}f_{z_2}$. Note that the root structural function ${}^{c_1}_{T}f_c$ of ${}_{T}f_{c_1}$ is not called anywhere, however, in order to be able to construct the result of ${}_{T}f_{z_1}$, ${}_{T}f_{c_1}$ should also be evaluated, thus when we call ${}^{z_1}_{T}f_b$, implicitly we also call ${}^{c_1}_{T}f_c$.

The second condition is introduced so as to be able to simulate conditions if c_i then z_i;. The meaning of

if $e \in \tilde{X}^{a}_{\overset{c_1}{T} f_a}$ then delete $\overset{z_2}{T} f_a(e)$
else if true then delete $\overset{z_1}{T} f_a(e)$,

is that if edge $e = (u, a, v)$ of document tree t is processed by the a row of $\overset{c_1}{T} f_a$ ($e \in X^{a}_{\overset{c_1}{T} f_a}$) and this edge does not become unreachable in $U_f \sqcap t$ after the evaluation of n.i. conditions (this is denoted with ($e \in \tilde{X}^{a}_{\overset{c_1}{T} f_a}$)), then edge $((\overset{z_2}{T} f_a, u), a, (\overset{z_2}{T} f_c, v))$ ($\overset{z_2}{T} f_a$ called on e) should be deleted from $U_f \sqcap t$. Otherwise edge $((\overset{z_1}{T} f_a, u), a, (w, v))$ ($\overset{z_1}{T} f_a$ called on e) should be deleted. Since $\overset{c_1}{T} f_a$ represents the aim of the XPath$_0$ expression of c_1, if e is in $X^{a}_{\overset{c_1}{T} f_a}$, and the corresponding edge is kept, then c_1 is satisfied by the appropriate subgraph of e, hence this branch of the condition is to be executed. (Note that as a result of an $\{a : t\}$ singleton $\overset{z_2}{T} f_a$ calls $\overset{z_2}{T} f_c$, while $\overset{z_1}{T} f_a$ calls no other structural function, thus the corresponding node is linked to w in U_{f_T}. Consequently, edges $((\overset{z_2}{T} f_a, u), a, (\overset{z_2}{T} f_c, v))$, $((\overset{z_1}{T} f_a, u), a, (w, v))$ are surely in $U_f \sqcap t$.)

As we have already indicated the matching nodes of XP_1, XP_2 should be repeated, which is the a node in both XPgraphs. In the first case a is a child of b, hence according to the rules of repetition $\overset{z_1}{T} f_b$ should call itself in its b row. Note that b is also the aim node of XP_1, and the aim node should also be repeated. However, according to the rules of repetition, it is enough to call $\overset{z_1}{T} f_b$ only once.

In the second case $\overset{z_2}{T} f_a$ calls itself in the a row. Note that according to the rewriting rules of XPath$_0$ expressions, since it is the aim node, again, we should also repeat c.

In the general case in its / row ${}_{T_{st}} f$ calls all of the root structural functions of ${}_{T} f_{q_i}$-s, i.e., $\{\mathsf{result} : \{\overset{z_1}{T} f_{\sigma_1}, \ldots, \overset{z_k}{T} f_{\sigma_k}\}\}$ should be constructed on the right side of the / row, where $\overset{z_i}{T} f_{\sigma_i}$ denotes the root structural function of ${}_{T} f_{q_i}$ ($1 \le i \le k$). (Remember that ${}_{T} f_{q_i}$ represents condition *if* c_i *then* z_i of T.)

Furthermore, when we are to simulate conditions:

$$\text{if } c_1 \text{ then } z_1; \ldots \text{ if } c_k \text{ then } z_k;$$

then the following should be written:

if $e \in \tilde{X}^{}_{\overset{c_1}{T} f_a}$ then delete $\overset{z_2}{T} f_a(e), \ldots, \overset{z_k}{T} f_a(e)$
if $e \in \tilde{X}^{}_{\overset{c_2}{T} f_a}$ then delete $\overset{z_1}{T} f_a(e), \overset{z_3}{T} f_a(e) \ldots, \overset{z_k}{T} f_a(e)$
$\vdots \qquad\qquad \vdots$
if $e \in \tilde{X}^{}_{\overset{c_k}{T} f_a}$ then delete $\overset{z_1}{T} f_a(e), \overset{z_1}{T} f_a(e) \ldots, \overset{z_{k-1}}{T} f_a(e)$

Here, the evaluation ${}^{c_i}_{T}f_a$ is independent from the evaluation ${}^{z_1}_{T}f_a, \ldots, {}^{z_k}_{T}f_a$. Thus, first ${}^{c_i}_{T}f_a$-s should be evaluated and then only the remaining ${}^{z_1}_{T}f_a$ should be called on e $(1 \leq i \leq k)$.

Theorem 3 *Keeping the above notation and suppositions,* P_T *is equivalent with* f_T.

Proof. Let t be a document tree. Suppose that T is called on edges $\hat{e}_1, \ldots, \hat{e}_r$, for $\hat{e}_i$ the s^{th} condition is satisfied, and the (r, σ_s)-rule is called on edges $e_1, \ldots, e_k$. Remember that this rule constructs a single ψ edge. Hence as a result of $\hat{e}_i$ P constructs a node with k outgoing ψ edges. Denote ψ_{e_i} the ψ edge constructed on e_i $(1 \leq i \leq k)$. From Theorem 2 we know that when ${}_{T}f_{z_s}$ starts on $\hat{e}_i$ and it stops on $e_1, \ldots, e_k$. Furthermore, for each e_i it constructs a ψ edge $(1 \leq i \leq k)$. Hence, as a result $\hat{e}_i$ ${}_{T}f_{q_s}$ constructs a node with k outgoing ψ edges. Additionally, in both cases ψ_{e_i} precedes ψ_{e_j}, if e_i precedes e_j in document order. With this, the construction of the corresponding mapping ϕ (Definition 6) is trivial. ■

Rewriting of programs with several rules

Programs with a given variable assignment. Now, we consider an $XSLT_0$ program P with rules $T_1, \ldots, T_m$. Again, we assume that there is a given variable assignment ρ. We do not assume that z_i-s are single at-expressions, but for the sake of transparency, we suppose that they contain only one at-expression as a leaf label. This leaf will be referred as *at-leaf*. This means that in its i^{th} condition T_j may call at most one T_k $(1 \leq j, k \leq m)$. The $XPath_0$ expression of this at-expression is still denoted $xp_j(z_i)$ or $xp(z_i)$, if j is clear from the context. We also assume that the edge tests of aim nodes of $xp(z_i)$-s are all different from $*$-s (except for $(st, /)$-rule). From this, it follows that in all cases T_j in its i^{th} condition calls the same T_k.

To delineate the relationships among the templates, we define an auxiliary graph, the *precedence graph (of rules of* P*)*. Its nodes are labelled with T_i-s. There is an edge from T_i to T_j labelled with k, if the k^{th} condition of T_i calls T_j $(1 \leq i, j \leq m)$.

Example 11 *As an example of simulation, consider the following program* $P = (T_1, T_2, T_3)$ *and its rewriting. In* P T_2 *is called on* b *descendants of the document edge. It constructs a* b *edge and calls* T_3 *on* c *parents.* T_3 *constructs a tree* $\{a : \{b : \{\}\}@\{c : \{\}\}\}$ *and stops.*

$\mathtt{template}\ \mathsf{st}(/, \epsilon) \qquad (\mathsf{T}_1)$
$\quad \mathtt{return}$
$\quad\quad$ *if* true *then* $\{\mathsf{result} : \{\}\}$ *at-expr:* $\mathsf{m}(\mathtt{desc::b}, \epsilon, \epsilon)$;
$\mathtt{end}.$

$\mathtt{template}\ \mathsf{m}(\mathsf{b}, \epsilon) \qquad (\mathsf{T}_2)$
$\quad \mathtt{return}$
$\quad\quad$ *if* true *then* $\{\mathsf{b} : \{\}\}$ *at-expr:* $\mathsf{m}(\mathtt{par::c}, \epsilon, \epsilon)$;
$\mathtt{end}.$
$\mathtt{template}\ \mathsf{m}(\mathsf{c}, \epsilon) \qquad (\mathsf{T}_3)$
$\quad \mathtt{return}$
$\quad\quad$ *if* true *then* $\{\mathsf{a} : \{\mathsf{b} : \{\}\}@\{\mathsf{c} : \{\}\}\}$
$\mathtt{end}.$

$$
\begin{array}{lll}
{}_{\mathsf{T}_1}\mathsf{f}_{/}: & (\{/ : \mathsf{t}\}) & = \{\mathsf{result} : {}^{z_1}_{\mathsf{T}_1}\ \mathsf{f}_{\mathsf{b}}(\mathsf{t})\} \\
 & (\{* : \mathsf{t}\}) & = \{\} \\
 & & \\
{}^{z_1}_{\mathsf{T}_1}\mathsf{f}_{\mathsf{b}}: & (\{\mathsf{b} : \mathsf{t}\}) & = {}^{z_1}_{\mathsf{T}_1}\mathsf{f}_{\mathsf{b}}(\mathsf{t}) \\
 & (\{* : \mathsf{t}\}) & = {}^{z_1}_{\mathsf{T}_1}\mathsf{f}_{\mathsf{b}}(\mathsf{t}) \\
 & & \\
{}^{z_1}_{\mathsf{T}_2}\mathsf{f}_{\mathsf{c}}: & (\{\mathsf{c} : \mathsf{t}\}) & = \textit{if } \textit{n.i.}({}^{z_1}_{\mathsf{T}_2}\mathsf{f}_{\mathsf{b}}(\mathsf{t})) \textit{ then } \{\mathsf{b} : \{\}\}@{}^{z_1}_{\mathsf{T}_2}\mathsf{f}_{\mathsf{c}}(\mathsf{t}) \\
 & (\{* : \mathsf{t}\}) & = {}^{z_1}_{\mathsf{T}_2}\mathsf{f}_{\mathsf{c}}(\mathsf{t}) \\
 & & \\
{}^{z_1}_{\mathsf{T}_2}\mathsf{f}_{\mathsf{b}}: & (\{\mathsf{b} : \mathsf{t}\}) & = \{\psi : \{\}\} \\
 & (\{* : \mathsf{t}\}) & = \{\}, \qquad \mathsf{X}^{\mathsf{b}}_{{}^{z_1}_{\mathsf{T}_1}\mathsf{f}_{\mathsf{b}}} = \mathsf{X}^{\mathsf{b}}_{{}^{z_1}_{\mathsf{T}_2}\mathsf{f}_{\mathsf{b}}} \\
 & & \\
{}^{z_1}_{\mathsf{T}_3}\mathsf{f}_{\mathsf{c}}: & (\{\mathsf{c} : \mathsf{t}\}) & = \{\mathsf{a} : \{\mathsf{b} : \{\}\}@\{\mathsf{c} : \{\}\}\}@{}^{z_1}_{\mathsf{T}_3}\mathsf{f}_{\mathsf{c}}(\mathsf{t}) \\
 & (\{* : \mathsf{t}\}) & = {}^{z_1}_{\mathsf{T}_3}\mathsf{f}_{\mathsf{c}}(\mathsf{t}), \qquad \mathsf{X}^{\mathsf{c}}_{{}^{z_1}_{\mathsf{T}_2}\mathsf{f}_{\mathsf{c}}} = \mathsf{X}^{\mathsf{c}}_{{}^{z_1}_{\mathsf{T}_3}\mathsf{f}_{\mathsf{c}}}
\end{array}
$$

The rewriting works similarly as in the previous subsection. The main difference is that here, instead of a star of ψ edges an arbitrary document tree is constructed. In order to properly simulate this construction, first, we have to note that constructions should be accomplished, when the matching node is reached. In other words this means that in most cases the construction should take place in the μ_i row of ${}^{z_j}_{\mathsf{T}_i}\mathsf{f}_{\mu_i}$, where μ_i denotes the matching node of template T_i ($1 \leq i \leq m$) (here, we have also assumed that the j^{th} condition

is satisfied first). This is the case in T_1 and T_3 in our example, where the matching nodes are b and c.

It may happen, however, that $\overset{z_j}{T_i} f_{\mu_i}$ is called in the check of a not-isempty condition, thus its construction does not appear in the result. This is the case in T_2. It is easy to see now that in this case the first axis in $xp(z_j)$ is either par, or anc. (Remember that the j^{th} condition, *if* c_j *then* z_j, is considered as $\texttt{self::}\mu_i[c_j]/xp(z_j)$.) Denote u_1 the upper node of the first XPgraph of $\texttt{self::}\mu_i/xp(z_j)$. (This means that there is a directed path $w_1 \dots w_s$ in this XPgraph, where $w_1 = u_1$, $w_s = \mu_i$ and the edges are not labelled or they are labelled with par.) Construction should take place in the u_1 row of $\overset{z_j}{T_i} f_{u_1}$. More accurately, in the then-branch of $\overset{z_j}{T_i} f_{u_1}$.

Now, suppose that the construction takes place in the φ row of $\overset{z_j}{T_i} f_{\varphi}$ and let $\overset{z_j}{T_i} f_{\sigma}(t)$ be the structural function to be called there. Then (in the then-branch) we construct z_j changing its at-expression label to $\overset{z_j}{T_i} f_{\sigma}(t)$. In our example in T_1, this new forest is $\{\texttt{result} : \overset{z_1}{T_1} f_b(t)\}$. (In T_3 z_1 does not have any at-expression leaf label, hence it should be used without changes.)

Furthermore, in f_P we have to connect the structural recursions of (m, σ)-rules using the precedence graph. For edge (T_i, s, T_j) we add restriction, $X^{\alpha_s^i}_{f_{\alpha_s^i}} = X^{\mu_j}_{f_{\mu_j}}$ $(1 \le i, j \le m)$, here α_s^i denotes the aim node of $xp_i(z_s)$.

Note that f_P does not specify how the basic forests constructed by structural recursions of templates should be connected. Consequently, it does not guarantee any order among the basic forests. In what follows, we show how the output should be constructed and how the order of the basic forests should be defined. Meanwhile, we also establish the equivalence of P and f_P.

Instantiation of an (m, σ)-rule by another. As a first step, recall the semantics of $XSLT_0$. Remember that if (e, m, ρ) is a local configuration, then $e \in E.t$, m is a mode, ρ is a given variable assignment, and it shows that (m, σ)-rule T is to be applied on e (here $lab(e) = \sigma$ and the parameters of T are in the domain of ρ). Now, let T_j, T_k be (m_j, σ_j)-, (m_k, σ_k)-rules and $e^1, e^2 \in E.t$. We say that (T_j, e^1) *instantiates* (T_k, e^2) *in its* s^{th} *condition*, if there is a $\xi \in \mathcal{T}^{\Delta}(LC^*(t))$ ($LC^*(t)$ denotes sequences of local configurations) s.t. in a former step, location configuration (e^1, m_j, ρ) was substituted with the result of T_j called on e^1, fo, where the s^{th} condition of T_j was satisfied, and as a result we get ξ. Furthermore, in fo there is a leaf label (e^2, m_k, ρ), which is to be substituted with the result of T_k called on e^2 $(1 \le j, k \le m)$. Here, $lab(e^1) = \sigma_j$ and $lab(e^2) = \sigma_k$.

Belonging to the same calling of f_{T_j}. To catch this notion with struc-

tural functions we introduce two notions. Suppose that $G_{f_P,t}$ has already been constructed. For a moment, however, suppose that we rewrite the premises that have already been deleted. For edges $e^1, e^2 \in E.t$, $e^1 \in R_{reg}(\mu_j, f_{\mu_j})$, $e^2 \in R_{reg}(\alpha^j_s, f\alpha^j_s)$, we are to define and check when e^1, e^2 belong to the same call of f_{T_j}. Here, remember that $e^1 \in R_{reg}(\mu_j, f_{\mu_j})$, $e^2 \in R_{reg}(\alpha^j_s, f\alpha^j_s)$ mean that structural functions respectively representing the matching node of T_j and the aim node of its s^{th} condition are called successfully on e^1 and e^2. Intuitively, e^1 and e^2 belong to the same of f_{T_j}, if there is a path from e^1 to e^2 specified by $xp_j(z_s)$ in $G_{f_P,t}$ with the rewritten premises, which shows that T_j was instantiated on e^1, and e^2 was selected for further processing.

Formally, consider the XPgraph of the skeleton of the s^{th} condition of T_j. Suppose that $u_1, \ldots, u_r$ are the upper nodes. Suppose also that each XP^u_i has two leaves v_{2i-1}, v_{2i} $(1 \le i \le r)$ (Fig. 10(d)). Denote $e_{v_i} \in E.U_{f_P}$ the edge corresponding to the then-branch of the v_i row of f_{v_i} $(1 \le i \le 2r)$. e^1 and e^2 *belong to the same call of* f_{T_j}, in notation $(f_{T_i}(e^1, e^2))$, if there exist edges in $E.t$ $e_0, \ldots, e_r$ s.t. $e_0 = e^1$, $e_r = e^2$, (i) $e_i \in R_{reg}(v_{2i}, f_{2i})$, and $e_i \in R_{reg}(v_{2i+1}, f_{2i+1})$ $(1 \le i \le r-1)$. (ii) for $(e_{v_{2j+1}}, e_j), (e_{v_{2j+2}}, e_{j+1}) \in E.U_{f_P} \sqcap t$, $(e_{v_{2j+2}}, e_{j+1})$ is reachable from $(e_{v_{2j+1}}, e_j)$ through a path containing exactly one neighbouring premise and then-edge pairs (the ancestor images of these conditional edges correspond to the u_{2j+1} row of $f_{u_{2j+1}}$) $(0 \le j \le r)$. (Consider Fig. 10(d) again.) Here, (i) means that e_i satisfies the corresponding register restriction $X^{v_{2i}}_{f_{v_{2i}}} = X^{v_{2i+1}}_{f_{v_{2i+1}}}$. On the other hand, condition (ii) says that e_j, e_{j+1} "correspond" to the two leaves of XP^{u_j+1}.

Note that, an algorithm that takes T_i and edges $e^1 \in R(\mu_i, f_{\mu_i})$ as input and finds all those edges e^2 to which $(f_{T_i}(e^1, e^2))$ holds, uses only the edges of paths from $(e_{v_{2j+1}}, e_j)$ to $(e_{v_{2j+2}}, e_{j+1})$ of condition (ii). Thus, it is possible to develop such an algorithm working in $O(|t||f|)$ time.

Instantiation of structural functions. With $Inst(f_{T_i}, e^1, s, f_{T_j}, e^2)$ we denote that $e^1 \in R_{reg}(\mu_i, f_{\mu_i})$, $e^2 \in R(\alpha^i_s, f\alpha^i_s)$, $e^2 \in R_{reg}(\mu_j, f_{\mu_j})$, and $(f_{T_i}(e^1, e^2))$. Clearly, with this definition we are to simulate the instantiation of T_j by T_i in the s^{th} condition. Note that, here e^2 also satisfies restriction $X^{\alpha^i_s}_{f_{\alpha^i_s}} = X^{\mu_j}_{f_{\mu_j}}$.

Note also that, when for e^1 we have found an edge e^2 s.t. $f_{T_i}(e^1, e^2)$, then the appropriate T_j, to which $Inst(f_{T_i}, e^1, s, f_{T_j}, e^2)$ holds, can be found using the precedence graph.

In what follows, the template to be called on the document edge is denoted T_{st}.

Definition 8 *Keeping the above notations, we say that* f_{T_r} is called on e via the document edge (in the k^{th} step)*, if there exists a sequence of edges of* E.t $e_1, \ldots, e_k$*, and* $f_{T_{i_1}}, \ldots, f_{T_{i_k}}$*, s.t.* $f_{T_{i_1}}$*,* $f_{T_{i_k}}$ *are respectively the same as* $f_{T_{st}}, f_{T_r}$*, and* $\mathtt{Inst}(f_{T_{i_j}}, e^j, s_j, f_{T_{i_{j+1}}}, e^{j+1})$ $(1 \leq j \leq k-1)$.

Note that register restrictions connecting matching nodes and aim nodes of different templates guarantee that if a subgraph of the input is processed by f_{T_i}, then the result is only included in the output, if f_{T_i} is called via the document edge.

Lemma 5 *For given XSLT$_0$ program* $P = (T_1, \ldots, T_m)$*, document tree* t*,* $e^1, e^2 \in$ E.t*,* (T_i, e^1) *instantiates* (T_j, e^2) *in its* s^{th} *condition iff* f_{T_i} *is called on* e^1 *via the document edge, and* $\mathtt{Inst}(f_{T_i}, e^1, s, f_{T_j}, e^2)$.

Proof. We use induction on the number of steps k in which f_{T_i} has been called on e^1 via the document edge. First suppose that $k = 0$, i.e., f_{T_i} is $f_{T_{st}}$ and e^1 is the document edge.

$\Rightarrow$: Clearly, in this case f_{T_i} is called on e^1 via the document edge. Suppose now that T_i, T_j are (m_i, σ_i)-, (m_j, σ_j)-rules. Then (T_i, e^1) instantiates (T_j, e^2) in its s^{th} condition, if there is a $\xi \in \mathcal{T}^{\Delta}(LC^*(t))$ s.t. in a former step, location configuration (e^1, m_i, ρ) was substituted with the result, fo, of the s^th condition of T_i called on e^1. Furthermore, in fo there is a leaf label (e^2, m_j, ρ), which is to be substituted with the result of T_j called on e^2 $(1 \leq j, k \leq m)$.

The fact that T_i has been called on e^1 means that $e^1 \in R(\mu_i, f_{\mu_i})$. Note that, since this instantiation of T_i does not depend on any other instantiation of T_j-s, here e^1 should not satisfy any rule register restrictions $(1 \leq j \leq m)$. Hence $e^1 \in R_{reg}(\mu_i, f_{\mu_i})$. Since the s^{th} condition is of the form:

$$q_s = \mathtt{self::}\sigma\mathtt{[}c_s\mathtt{]/}xp_i(z_s),$$

the fact that the s^{th} condition of T_i has been satisfied guarantees that $e^2 \in q_s(e^1)$ (Theorem 2), which means that $e^2 \in R(\alpha_s^i, f_{\alpha_s^i})$. Since T_j is called on e^2, we know that $e^2 \in R(\mu_j, f_{\mu_j})$. It is also obvious that $(f_{T_i}(e^1, e^2))$. Consequently $\mathtt{Inst}(f_{T_i}, e^1, s, f_{T_j}, e^2)$. Furthermore, it has also turned out that f_{τ_j} is called on e^2 via the root.

The general step of this direction is similar.

$\Leftarrow$: The proof is similar to the proof of the other direction. In this case, we only have to change the "direction" of the reasoning. ■

Connection of the basic forests. Finally, we should connect the basic forests in an order corresponding to the order given by the simulated XSLT$_0$

program. Here, the fact that $\mathtt{Inst}(f_{T_i}, e^1, s, f_{T_j}, e^2)$ holds means that a basic forest fo belonging to the s^{th} condition of T_i is constructed. Label its root and at-leaf with $(f_{T_i}, e^1), (f_{T_j}, e^2)$ respectively. (Remember that the at-leaf is the leaf previously labelled by an at-expression.) The result forest is denoted $fo(f_{T_i}, e^1)$. Afterwards, we connect the basic forests with ε edges. Namely, we add ε edges from leaf labels of (f_{T_j}, e^2) to root labels (f_{T_j}, e^2). At the end these ε edges should be eliminated. Note that, with the connection of the at-leaf (f_{T_j}, e^2) of $fo(f_{T_i}, e^1)$ and the root of $fo(f_{T_j}, e^2)$, we simulate that moment, when in the instantiation of T_j by T_i, the leaf with location configuration label (e^2, m_j, ρ) is substituted with the result of T_j called on e^2.

Order of the result. In order to define an order in the result, we first give an order among the basic forests. We construct an auxiliary graph, $\mathtt{Inst}_{P,t}$, whose nodes are labelled with (f_{T_i}, e)-s. We add an edge from (f_{T_i}, e^1) to (f_{T_j}, e^2) with label s, if $\mathtt{Inst}(f_{T_i}, e^1, s, f_{T_j}, e^2)$.

Now, $fo(f_{T_i}, e^1)$ precedes $fo(f_{T_j}, e^2)$, (i) if (f_{T_i}, e^1) and (f_{T_j}, e^2) have a common parent (f_{T_s}, e) in $\mathtt{Inst}_{P,t}$, and e^1 precedes e^2 in the document order $(1 \leq i, j \leq m)$. Clearly, this case represents that, when T_s has been instantiated on e and both e^1 and e^2 have been chosen for further processing. Thus e^1 and e^2 have been selected by the same XPath_0 expression and in the same mode, consequently $i = j$ holds.

(ii) Denote (f_{T_s}, e) the first common ancestor of (f_{T_i}, e^1) and (f_{T_j}, e^2) in $\mathtt{Inst}_{P,t}$. Suppose that (f_{T_k}, e^3), (f_{T_k}, e^4) are children of (f_{T_s}, e), and (f_{T_i}, e^1), (f_{T_j}, e^2) are reachable through (f_{T_k}, e^3), (f_{T_k}, e^4) respectively. Then $fo(f_{T_i}, e^1)$ precedes $fo(f_{T_j}, e^2)$, if $fo(f_{T_k}, e^3)$ precedes $fo(f_{T_k}, e^4)$.

Now, if $fo(f_{T_i}, e^1)$ precedes $fo(f_{T_j}, e^2)$, then the edges of $fo(f_{T_i}, e^1)$ precedes the edges of $fo(f_{T_j}, e^2)$. The order of the edges in $fo(f_{T_i}, e^1)$ is given by the corresponding ssd-expression.

Theorem 4 *Let* P *an XSLT*$_0$ *program without variables, then* f_P *is equivalent with* P.

Proof. Let t be a document tree. Clearly, the construction of $\tau_P(t)$ can be described as a sequence of (m, σ)-rules $T_{i_1}, \ldots, T_{i_k}$ s.t. in the j^{th} step T_{i_j} instantiates $T_{i_{j+1}}$ $(1 \leq j \leq k-1)$. Here, T_{i_1} is the $(st, /)$-rule, and there may exist several such sequences for the same construction of $\tau_P(t)$, but they are all of the same length.

Now, we prove the theorem using induction on this length. If $k = 2$, the statement follows from Lemma 5.

Next, suppose that the statement holds for all document trees, where the

aforementioned sequences have length $\leq k$. Let t be such a document tree, where this length is $k+1$. Suppose that, in the last step $T_{i_{k+1}}$ is called on location configuration (e, m, ρ). Trivially, if the corresponding leaf did not have any location configuration label, i.e., the construction stopped before the call of $T_{i_{k+1}}$, then, according to our assumptions, the appropriate mapping ϕ^1 (Definition 6) would exist between $V.\tau_P(t)$ and $V.f_P(t)$. We also know that there is an edge e' and number s s.t. $Inst(f_{T_{i_k}}, e', s, f_{T_{i_{k+1}}}, e)$. Furthermore, using again Lemma 5, the result, t_T, of $T_{i_{k+1}}$ called on e is equivalent with the result, t_f, of $f_{T_{i_{k+1}}}$ called also on e ($e \in R(\tau_m^{i_{k+1}}, f_{\tau_m^{i_{k+1}}})$). Hence, the appropriate mapping ϕ^2 can be given between $V.t_T$ and $V.t_f$.

Now, we only have to extend ϕ^1 with ϕ^2. Denote ϕ this new mapping. Obviously, condition (i)-(ii) of Definition 6 holds for ϕ. The truthfulness of condition (iii) can be proven easily with the use of the rules of defining an order among the connected basic forests. ■

Infinite loops. It is not difficult to see that how our method avoids infinite loops. Consider the following program P and its rewriting f_P:

```
template st(/, ϵ)        (T1)
    return
        if c1 = true then {c : {}}; at-expr: m(child::*, ϵ, ϵ)
end.

template m(a, ϵ)         (T2)
    return
        if c1 = true then {a : {}}; at-expr: m(desc::b, ϵ, ϵ)
end.

template m(b, ϵ)         (T3)
    return
        if c1 = true then {b : {}}; at-expr: m(anc::a, ϵ, ϵ)
end.
```

$$ {}^{z^1}_{T_2}f_a\colon (\{a:t\}) = \{a :^{z^1}_{T_2} f_b(t)\}@^{z^1}_{T_2}f_a(t) \qquad (\{*:t\}) = {}^{z^1}_{T_2}f_a(t) $$

$$ {}^{z^1}_{T_2}f_b\colon (\{b:t\}) = {}^{z^1}_{T_2}f_b(t) \qquad (\{*:t\}) = {}^{z^1}_{T_2}f_b(t) $$

$$ {}^{z^1}_{T_3}f_a\colon (\{a:t\}) = \text{if n.i.}({}^{z^1}_{T_3}f_b(t)) \text{ then } \{b:\{\}\}@{}^{z^1}_{T_3}f_a \text{ else } {}^{z^1}_{T_3}f_a(t) $$

$$ (\{*:t\}) = {}^{z^1}_{T_3}f_a(t) $$

$$\begin{array}{ll} {}^{z_1}_{T_3}f_b\colon & (\{b:t\}) = \{\psi:\{\}\} \\ & (\{*:t\}) = {}^{z_1}_{T_3}f_b(t) \end{array}$$

$${}_{T_2}f_s\colon\ (\{/:t\}) = \text{if } \mathtt{true} \text{ then } {}^{z_1}_{T_2}f_a(t) \qquad {}_{T_3}f_s\colon\ (\{/:t\}) = \text{if } \mathtt{true} \text{ then } {}^{z_1}_{T_3}f_a(t)$$

$${}_Pf_s\colon\ \ (\{/:t\}) = \{/:({}_{T_2}f_s(\{/:t\}), {}_{T_3}f_s(\{/:t\}))\},$$

$$X^a_{{}^{z_1}_{T_2}f_a} = X^a_{{}^{z_1}_{T_3}f_a},\ X^b_{{}^{z_1}_{T_2}f_b} = X^b_{{}^{z_1}_{T_3}f_b}$$

Clearly, for document trees t with root edge a having a b descendant, P enters into an infinite loop. On the other hand, as f_P traverses t top-down, every edge is "considered" at most once. If an a edge has a b descendant, then ${}^{z_1}_{T_2}f_a$ constructs an a edge, to which a b edge is connected constructed by ${}^{z_1}_{T_3}f_a$.

How variable assignment can be obtained. Till now, we have always assumed the existence of a given variable assignment. In what follows, we are to show, how this assignment, ρ, can be given. Remember that variable definitions are only allowed to appear in the $(st, /)$-rule. Consider now variable definition $x = r$. Here r is an XPath_0 expression. Denote f^{ro}_x the root structural function of the structural recursion f_x representing r. Then, for an XSLT_0 program P with templates $T_1, \ldots, T_m$, variables $x_1, \ldots, x_n$, the $\{/ : t\}$ row of ${}_Pf_s$ should extended with these root structural functions.

$$f_s : (\{/:t\}) = f^{ro}_{x_1}, \ldots, f^{ro}_{x_n}, \ldots, \{/:({}_{T_1}f_s(\{/:t\}), \ldots, {}_{T_1}f_s(\{/:t\}))\}$$

First, for a given document tree t, the results of f_{x_1} should be constructed. The edge-set on which f_{x_1} stops gives $\rho(x_1)$. Next, $\rho(x_2)$ is calculated. Here, we may have to use the result of $\rho(x_1)$. After the construction of $\rho(x_n)$ we get ρ, and the construction of $f_P(t)$ should be continued with this variable assignment. As an example, we give the rewriting of Example 1.

$$\begin{array}{lll} {}_{T_{st}}f_{/}: & (\{/:t\}) & = {}^{x}_{T_{st}}f_{gr}(t), \{\mathtt{result}:{}^{z_1}_{T_{st}}f_{gr}(t)\} \\ & (\{*:t\}) & = \{\} \end{array}$$

$$\begin{array}{lll} {}^{z_1}_{T_{st}}f_{gr}: & (\{\mathtt{group}:t\}) & = {}^{z_1}_{T_{st}}f_{gr}(t) \\ & (\{*:t\}) & = {}^{z_1}_{T_{st}}f_{gr}(t) \end{array}$$

$$
\begin{array}{lll}
{}_{T_{st}}{}^{x}f_{gr}: & (\{group : t\}) & = {}_{T_{st}}{}^{x}f_{id}(t) \quad {}_{T_{st}}{}^{x}f_{id}(t) : (\{id : t\}) = {}_{T_{st}}{}^{x}f_{id}(t) \\
 & (\{* : t\}) & = {}_{T_{st}}{}^{x}f_{gr}(t) \qquad\qquad\qquad (\{* : t\}) = {}_{T_{st}}{}^{x}f_{gr}(t)
\end{array}
$$

$$
\begin{array}{lll}
{}_{T_{st}}{}^{x}f^{pr}_{gr}: & (\{group : t\}) & = \text{if } n.i.({}_{T_{st}}{}^{x}f^{pr}_{emp}(t)) \text{ then } {}^{x}f^{pr}_{gr}(t) \\
 & & \qquad\qquad \text{else } {}_{T_{st}}{}^{x}f^{pr}_{gr}(t) \\
 & (\{* : t\}) & = {}_{T_{st}}{}^{x}f^{pr}_{gr}(t)
\end{array}
$$

$$
\begin{array}{lll}
{}_{T_{st}}{}^{x}f^{pr}_{emp}: & (\{emp : t\}) & = {}_{T_{st}}{}^{x}f^{pr}_{na}(t) \\
 & (\{* : t\}) & = \{\}
\end{array}
$$

$$
\begin{array}{lll}
{}_{T_{st}}{}^{x}f^{pr}_{na}: & (\{name : t\}) & = \text{if } val(name) = Ann \text{ then } \{\psi : \{\}\} \\
 & (\{* : t\}) & = \{\}, \quad X^{gr}_{{}_{T_{st}}{}^{x}f_{gr}} = X^{gr}_{{}_{T_{st}}{}^{x}f^{pr}_{gr}}
\end{array}
$$

$$
\begin{array}{lll}
{}^{c_1}_{T_1}f_{gr}: & (\{group : t\}) & = \text{if } n.i.({}^{c_1}_{T_1}f_{to}(t)) \text{ then } {}^{c_1}_{T_1}f_{gr}(t) \\
 & & \qquad\qquad \text{else } {}^{c_1}_{T_1}f_{gr}(t) \\
 & (\{* : t\}) & = {}^{c_1}_{T_1}f_{gr}(t)
\end{array}
$$

$$
\begin{array}{lll}
{}^{c_1}_{T_1}f_{to}: & (\{topMgr : t\}) & = {}^{c_1}_{T_1}f_{na}(t) \\
 & (\{* : t\}) & = \{\}
\end{array}
$$

$$
\begin{array}{lll}
{}^{c_1}_{T_1}f_{na}: & (\{name : t\}) & = \text{if } val(name) = John \text{ then } \{\psi : \{\}\} \\
 & (\{* : t\}) & = \{\}
\end{array}
$$

$$
\begin{array}{lll}
{}^{z_1}_{T_1}f_{gr}: & (\{group : t\}) & = \{topGroup : \{id : {}^{z_1}_{T_1}f^{val}_{id}(t)\}\}@{}^{z_1}_{T_1}f^{Ann}_{gr}(t)@{}^{z_1}_{T_1}f_{gr}(t) \\
 & (\{* : t\}) & = {}^{z_1}_{T_1}f_{gr}(t), \quad X^{gr}_{{}^{c_1}_{T_1}f_{gr}} = X^{gr}_{{}^{z_1}_{T_1}f_{gr}}, X^{gr}_{{}^{z_1}_{T_{st}}f_{gr}} = X^{gr}_{{}^{z_1}_{T_1}f_{gr}}
\end{array}
$$

$$
\begin{array}{lll}
{}^{z_1}_{T_1}f^{val}_{id}: & (\{id : t\}) & = \{\} \quad {}^{z_1}_{T_1}f^{Ann}_{gr} : (\{group : t\}) = {}^{z_1}_{T_1}f^{Ann}_{gr}(t) \\
 & (\{* : t\}) & = \{\} \qquad\qquad (\{* : t\}) \qquad = \{\}
\end{array}
$$

$$
\begin{array}{lll}
{}^{c_1}_{T_2}f_{gr}: & (\{group : t\}) & = \text{if } n.i.({}^{c_1}_{T_2}f_{id}(t)) \text{ then } {}^{c_1}_{T_2}f_{gr}(t) \\
 & & \qquad\qquad \text{else } {}^{c_1}_{T_2}f_{gr}(t) \\
 & (\{* : t\}) & = {}^{c_1}_{T_2}f_{gr}(t)
\end{array}
$$

$$
\begin{array}{lll}
{}^{c_1}_{T_2}f_{id}: & (\{id : t\}) & = \text{if } val(id) = x \text{ then } \{\psi : \{\}\} \\
 & (\{* : t\}) & = \{\}
\end{array}
$$

$$
\begin{array}{lll}
{}^{z_1}_{T_2}f_{gr}: & (\{group:t\}) & = \{id :^{z_1}_{T_2} f^{val}_{id}(t)\} \\
 & (\{*:t\}) & = {}^{z_1}_{T_1}f_{gr}(t), \quad X^{gr}_{c\ {}^{z_1}_{T_2}f_{gr}} = X^{gr}_{{}^{z_1}_{T_2}f_{gr}}, X^{gr}_{{}^{z_1}_{T_{st}}f_{gr}} = X^{gr}_{{}^{z_1}_{T_2}f_{gr}}
\end{array}
$$

$$
\begin{array}{lll}
{}^{z_1}_{T_2}f^{val}_{id}: & (\{id:t\}) & = \{\} \quad {}^{z_1}_{T_3}f_{id}:(\{id:t\}) = \{val(id):\{\}\}@{}^{z_1}_{T_3}f_{id}(t) \\
 & (\{*:t\}) & = \{\} \qquad (\{*:t\}) = {}^{z_1}_{T_3}f_{id}
\end{array}
$$

6 Conclusions

In this paper we have introduced a new version of structural recursions, where we have added registers to be able to connect the results of structural functions called on the same XML document. To underpin the usefulness of this extension, we have showed how a practically important fragment of XPath and XSLT can be implemented with these structural recursions. As it has turned out, our technique has the same efficiency as the fastest implementation algorithm [8] known by the authors of this paper.

In the near future we shall work out how schema information given in the form of extended DTD-s [12] can be incorporated into our model. Looking for further optimization possibilities, we also plan to implement our techniques in a software and compare its speed with the existing XPath and XSLT implementations.

References

[1] S. Abiteboul, P. Buneman, D. Suciu, *Data on the Web: From Relations to Semistructured Data and XML*, Morgan Kaufmann Publishers, 2000. ⇒ 167

[2] A. Benczúr, B. Kósa, Static Analysis of Structural Recursion in Semistructured Databases and Its Consequences. *Advances in Databases and Information Systems, 8th East European Conference Proceedings*, 2004, pp. 189–203. ⇒ 166, 179

[3] A. Benczúr, B. Kósa, Satisfiability and Containment Problem of Structural Recursions with Conditions with Respect to XML. *To appear.* ⇒ 166

[4] A. Benczúr, A. Kiss, B. Kósa, Implementation of XPath Using Structural Recursions, *Advances in Databases and Information Systems, 13th East European Conference. To appear.* ⇒ 196

[5] G. J. Bex, S. Maneth, F. Neven, A Formal Model for an Expressive Fragment of XSLT. 2000. *Manuscript.* ⇒ 166, 167, 170, 171, 172

[6] V. Breazu-Tannen, P. S. Buneman, Structural Recursion as a Query Language, *Proceedings of the 3rd International Workshop on Database Programming Languages*, 1991, pp. 9–19. ⇒ 166

[7] P. Buneman, M. Fernandez, D. Suciu, UnQL: a query language and algebra for semistructured data based on structured recursion. *The VLDB Journal*, (2000) 76–110. ⇒ 166, 167

[8] G. Gottlob, C. Koch, R. Pichler, Efficient Algorithms for Processing XPath Queries, *Proceedings of the 28th International Conference on Very Large Data Bases*, 2002. ⇒ 166, 211

[9] G. Gottlob, C. Koch, R. Pichler, XPath Processing in a Nutshell. *ACM SIGMOD Record*, **32**, 2 (2003) 21–27. ⇒ 167, 169

[10] G. Gottlob, C. Koch, R. Pichler, XPath Query Evaluation: Improving Time and Space Efficiency, *Proceedings of the 19th IEEE International Conference on Data Engineering*, 2003.

[11] W. Martens, F. Neven: On the complexity of typechecking top-down XML transformations, *Theoret. Comput. Science*, **336**, 1 (2005) 153–180. ⇒ 166

[12] W. Martens, F. Neven, T. Schwentick, G. J. Bex: Expressiveness and complexity of XML Schema. *ACM Transactions on Database Systems*, **31**, 3 (2006) 770–813. ⇒ 211

[13] T. Milo, D. Suciu, V. Vianu. Type checking for XML transformers, *Proceedings of the Nineteenth ACM Symposium on Principles of Database Systems*, 2000, pp. 11–22.

[14] F. Neven. Automata, Logic and XML. *Proceedings of the 16th International Workshop and 11th Annual Conference of the EACSL on Computer Science Logic*, 2002, pp. 2–26. ⇒ 165, 167

[15] Word Wide Web Consortium. Extensible Markup Language (XML) 1.1, 2004. http://www.w3.org/TR/2004/REC-xml11-20040204/ ⇒ 166, 167, 168

[16] Word Wide Web Consortium. XML Path Language (XPath) Version 1.0, 1999. http://www.w3.org/TR/1999/REC-xpath-19991116/ ⇒ 165, 166, 168, 169, 170

[17] Word Wide Web Consortium. XSL Transformations (XSLT) Version 1.0, 1999. http://www.w3.org/TR/1999/REC-xslt-19991116/ ⇒ 165, 166, 170, 175

[18] Word Wide Web Consortium. XML Path Language (XPath) 2.0, 2007. http://www.w3.org/TR/2007/REC-xpath20-20070123/ ⇒ 168

[19] Word Wide Web Consortium. XSL Transformations (XSLT) Version 2.0, 2007. http://www.w3.org/TR/2007/REC-xslt20-20070123/

Received: April 26, 2009