University of Pennsylvania
ScholarlyCommons

Technical Reports (CIS) | Department of Computer & Information Science

October 2004

Paths Into Patterns

Vladimir Gapeyev
University of Pennsylvania

Benjamin C. Pierce
University of Pennsylvania, bcpierce@cis.upenn.edu

Follow this and additional works at: http://repository.upenn.edu/cis_reports

Recommended Citation
Gapeyev, Vladimir and Pierce, Benjamin C., "Paths Into Patterns" (2004). *Technical Reports (CIS)*. Paper 26.
http://repository.upenn.edu/cis_reports/26

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-04-25.

This paper is posted at ScholarlyCommons. http://repository.upenn.edu/cis_reports/26
For more information, please contact repository@pobox.upenn.edu.

Paths Into Patterns

Abstract
The XML Path Language (XPATH) is an industry standard notation for addressing parts of an XML document. It is supported by many XML processing libraries and has been used as the foundation for several dedicated XML processing languages. Regular patterns, an alternative way of investigating and destructing XML documents, were first proposed in the XDUCE language and feature in a number of its descendants.

The processing styles offered by XPATH and by regular patterns are each quite convenient for certain sorts of tasks, and the designer of a future XML processing language might well like to provide both. This designer might wonder, however, to what extent these mechanisms can be based on a common foundation. Can one be implemented by translating it into the other? Can aspects of both be combined into a single notation?

As a first step toward addressing these questions, we show in this paper that a language closely related to the "downward axis" fragment of XPATH can be accurately translated into ambiguous XDUCE-style regular patterns with a "collect all matches" interpretation.

Comments
University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-04-25.

This technical report is available at ScholarlyCommons: http://repository.upenn.edu/cis_reports/26

Paths Into Patterns

Vladimir Gapeyev Benjamin C. Pierce

Technical Report MS-CIS-04-25
Department of Computer and Information Science
University of Pennsylvania

October 27, 2004

Abstract

The XML Path Language (XPath) is an industry standard notation for addressing parts of an XML document. It is supported by many XML processing libraries and has been used as the foundation for several dedicated XML processing languages. Regular patterns, an alternative way of investigating and destructing XML documents, were first proposed in the XDuce language and feature in a number of its descendants.

The processing styles offered by XPath and by regular patterns are each quite convenient for certain sorts of tasks, and the designer of a future XML processing language might well like to provide both. This designer might wonder, however, to what extent these mechanisms can be based on a common foundation. Can one be implemented by translating it into the other? Can aspects of both be combined into a single notation?

As a first step toward addressing these questions, we show in this paper that a language closely related to the "downward axis" fragment of XPath can be accurately translated into ambiguous XDuce-style regular patterns with a "collect all matches" interpretation.

1 Introduction

An XPath [16] expression specifies how to traverse an XML value, following one of several possible "axes" at each step and performing tests on node labels and types to eventually reach a set of matching nodes. For example, evaluating the path expression

```
child::person / child::name / child::first
```

against the XML value

```
<contacts>
    <person>
        <name><first>Haruo</first> <last>Hosoya</last></name>
        <email>hahasoya</email>
    </person>
    <person>
        <name><first>Jerome</first> <last>Vouillon</last></name>
        <tel>123</tel>
    </person>
  </contacts>
```

yields the nodes `<first>Haruo</first>` and `<first>Jerome</first>`.

XPath is supported by many XML processing libraries and has been used as the foundation for several dedicated XML processing languages, notably XSLT [14] and XQuery [19].

Regular patterns, an alternative way of investigating and destructing XML documents, were first proposed in the XDuce language [8] and feature in a number of its descendants, including Xtatic [5, 4], CDuce [1], XHaskell [10], and Broberg, Farre, and Svenningsson's Haskell extension [2]. A regular pattern is just a *regular type* [9] decorated with some variable binders. For example, the regular pattern[1]

```
contacts{ person{name{first{any} as f, last{any} as x}}, any }
```

can be matched against the same XML document as above to yield the environment mapping the pattern variable `f` to the tree `<first>Haruo</first>` and the variable `x` to `<last>Hosoya</last>`. The sub-pattern inside the `contacts` node describes a *sequence* of sub-trees, where the first must match the sub-pattern `person{name{first{any} as f, last{any} as x}}` (this is why the pattern retrieves the information for `Hosoya` rather than `Vouillon`) and the rest must match the wildcard pattern `any`.

Different dialects of XDuce (there have been several) and various descendants have advocated different decisions about what to do with *ambiguous* patterns such as this variant of the above example (note that the `person` node is both followed *and preceded* by `any`):

```
contacts{any, person{name{first{any} as f, last{any} as x}}, any}
```

Some use a "first match" semantics, others "longest match." Yet others give a non-deterministic semantics, promising to return *some* match, but refusing to say which. Some refuse to assign any meaning at all to such patterns.

Still another useful alternative is to ask that an ambiguous regular pattern collect up *all* matching sub-trees so that they can be iterated over, returned as a sequence, etc. Besides being useful in itself, this interpretation of regular patterns raises an interesting linguistic question: What is the precise relation between these two—superficially quite different—notations for extracting sets of subtrees from XML trees, path expressions and ambiguous regular patterns? To what extent can each be encoded in the other? Is there some natural generalization or hybrid that combines elements of both?

In this paper, we take a first step toward addressing these questions. We define a simplified language of path expressions closely related to the "downward axis" fragment of XPath and demonstrate that this language can be translated into ambiguous XDuce-style regular patterns.

[1] Our notation differs from XDuce slightly here: since square brackets are the standard notation for predicates in XPath expressions, we use curly braces for labeled tree patterns and values.

Following some notational preliminaries in Section 2, we define the syntax and semantics of regular patterns in Section 3 and of our path expression language in Section 4. Section 5 presents the translation from the latter to the former, sketches a proof of its correctness, and shows that the pattern type inference mechanism used in XDuce can be used to give semantically precise types for translated path expressions. Sections 6 and 7 discuss related and future work.

2 Preliminary Definitions

Sequence values and *trees* are defined recursively by the grammar

$$\begin{array}{lcll} v & ::= & t_1, \ldots, t_m & \text{sequence, } 0 \leq m \\ t & ::= & l\{v\} & \text{tree} \end{array}$$

where l ranges over a countable set of *labels*. Sequence values are ranged over by variables v, w and the set of all values is denoted *Val*. Trees are ranged over by variables t, u, and *Tree* stands for the set of all trees. The empty sequence (when $m = 0$) is written $()$. Given two sequences v and w their concatenation is denoted v, w.

Pattern variables (or just *variables*) are ranged over by x, y, z, possibly with subscripts. *Var* denotes the set of all variables. An *environment* Σ is a mapping from a finite subset $dom(\Sigma)$ of *Var* to *Val*. The set of all environments is denoted *Env*. A concrete environment Σ is specified by writing down the mappings for all its variables: $\Sigma = \{x_1 = v_1, x_2 = v_2, \ldots, x_n = v_n\}$. In particular, the environment Σ with $dom(\Sigma) = \emptyset$ is written $\{\}$. If two environments Σ_1, Σ_2 are such that $dom(\Sigma_1) \cap dom(\Sigma_2) = \emptyset$, they can be combined, in the standard way, into the environment Σ_1, Σ_2 that contains the bindings of both.

A *relation* between sets A and B is a set $R \subseteq A \times B$. The fact that $(a, b) \in R$ is also written in infix form $a\ R\ b$. For two relations $R_1 \subseteq A \times B$ and $R_2 \subseteq B \times C$, their *composition* is the relation $R_1 \circ R_2 \subseteq A \times C$ defined as $R_1 \circ R_2 = \{\, (a, c) \in A \times C \mid \exists\, b \in B.\ a\ R_1\ b \text{ and } b\ R_2\ c \,\}$.

3 Patterns

This section recaps basic definitions of regular tree pattern matching from XDuce [8], in the notation used in this paper. The only significant difference from XDuce is that we take the intersection of two patterns as a syntactic pattern constructor, as in CDuce [1].

The syntax of patterns (which are ranged over by P, R) is defined by the following grammar, where x and l range over variables and labels, as introduced in Section 2, and X ranges over a countable set of *pattern names*.

$$\begin{array}{lcll} P & ::= & & \textit{patterns} \\ & \mid & L\{P\} & \text{tree} \\ & \mid & () & \text{empty sequence} \\ & \mid & P_1, P_2 & \text{concatenation} \\ & \mid & P_1 \cup P_2 & \text{union} \\ & \mid & P_1 \cap P_2 & \text{intersection} \\ & \mid & P \ \mathsf{as}\ x & \text{binding} \\ & \mid & X & \text{pattern name} \\ \\ L & ::= & & \textit{label patterns} \\ & \mid & l & \text{atomic label} \\ & \mid & * & \text{label wildcard} \end{array}$$

Parentheses can be freely used to specify order of pattern operators, as in $(l\{P_1\}, P_2) \cup ()$.

A *pattern definition table* (or just *table*) is a mapping D from a finite set of pattern names to patterns such that, for any $X \in dom(D)$, if a name X' appears in $D(X)$, then also $X' \in dom(D)$. If $D(X) = P$, we also say that the definition, or equation, $X = P$ is in D. Note that tables permit recursion—when a name

refers to itself, possibly via a "loop" of pattern definitions. A table D' *extends* table D if D' extends D as a function, i.e. $D' \supseteq D$.

The *standard table* is the table D_{std} that consists of the equations

$$\begin{array}{lcl} \textsc{anyone} & = & \texttt{*\{}\textsc{any}\texttt{\}} \\ \textsc{any} & = & (\textsc{anyone}, \textsc{any}) \cup (). \end{array}$$

In the following we always assume that any table D under consideration extends the standard table D_{std}.

A pattern P is *defined* under table D, or D *defines* P, if for every pattern name X appearing in P, we have $X \in \mathit{dom}(D)$. This implies that every name X', accessible from P by recursively following the definitions, is also in $\mathit{dom}(D)$.

We also extend the last definition to handle derived pattern forms P^*, P^+ and $P?$. Namely, it is said that P^* is defined under table D if P is defined under D, and D contains the equation

$$D(\textsc{star}^P) = P, \textsc{star}^P \cup ().$$

That is, whenever we mention a pattern P^*, we actually mean the pattern $\textsc{star}^P$. Then P^+ stands for P, P^*, and $P?$ is short for $P \cup ()$.

Following XDuce, we assume a couple syntactic restrictions on patterns, or, more precisely, on pattern definition tables (formal details can be found in [9] and [8]):

- Regularity: all "loops" through the definitions in a table D from a variable X back to itself must pass through the body P of at least one tree pattern $L\texttt{\{}P\texttt{\}}$. This ensures that the value sets corresponding to patterns, according to the semantics to be defined shortly, are regular tree languages.
- Linearity: no variable x is bound in a pattern twice, except in union patterns, where branches must bind exactly the same variables. (This notion also takes into account not just a syntactic pattern, but its "unfolding" w.r.t. its defining table D.)

The set of *free variables* of pattern P under definitions D is denoted $\mathit{fv}_D(P)$ and consists of all variables x occurring in P and, recursively, in any patterns reachable from P in D. Formally, fv_D is the function which is the least fixed point of the following system of monotone equations:

$$\begin{array}{lcl} \mathit{fv}_D(L\texttt{\{}P\texttt{\}}) & = & \mathit{fv}_D(P) \\ \mathit{fv}_D(()) & = & \emptyset \\ \mathit{fv}_D(P_1, P_2) & = & \mathit{fv}_D(P_1) \cup \mathit{fv}_D(P_2) \\ \mathit{fv}_D(P_1 \cup P_2) & = & \mathit{fv}_D(P_1) \cup \mathit{fv}_D(P_2) \\ \mathit{fv}_D(P_1 \cap P_2) & = & \mathit{fv}_D(P_1) \cup \mathit{fv}_D(P_2) \\ \mathit{fv}_D(X) & = & \mathit{fv}_D(D(X)) \\ \mathit{fv}_D(P \ \mathsf{as}\ x) & = & \mathit{fv}_D(P) \cup \{x\} \end{array}$$

In most cases D is clear from context, and we write just $\mathit{fv}(P)$ instead of $\mathit{fv}_D(P)$.

If a pattern P is defined under a table D, then its semantics is a relation $[\![P]\!]_D \subseteq \mathit{Val} \times \mathit{Env}$ defined by the following inference rules. If $v\ [\![P]\!]_D\ \Sigma$, we say that matching value v against pattern P (which is defined under table D) results in environment Σ. When table D is clear from the context, we elide it and write just $v\ [\![P]\!]\ \Sigma$. In particular, the following rules defining the relation do not change D (and only one rule actually references D), so we use the light notation assuming a fixed table D is given.

$$\frac{}{()\ [\![()]\!]\ \{\}} \qquad \frac{l \in L \qquad v\ [\![P]\!]\ \Sigma}{l\texttt{\{}v\texttt{\}}\ [\![L\texttt{\{}P\texttt{\}}]\!]\ \Sigma} \qquad \frac{}{l \in l} \qquad \frac{}{l \in \texttt{*}}$$

$$\frac{v = v_1, v_2 \qquad v_1\ [\![P_1]\!]\ \Sigma_1 \qquad v_2\ [\![P_2]\!]\ \Sigma_2}{v\ [\![P_1, P_2]\!]\ \Sigma_1, \Sigma_2} \qquad \frac{v\ [\![P_1]\!]\ \Sigma_1}{v\ [\![P_1 \cup P_2]\!]\ \Sigma_1} \qquad \frac{v\ [\![P_2]\!]\ \Sigma_2}{v\ [\![P_2 \cup P_2]\!]\ \Sigma_2}$$

$$\frac{v\ [\![P_1]\!]\ \Sigma_1 \qquad v\ [\![P_2]\!]\ \Sigma_2}{v\ [\![P_1 \cap P_2]\!]\ \Sigma_1, \Sigma_2} \qquad \frac{v\ [\![P]\!]\ \Sigma}{v\ [\![P\ \mathsf{as}\ x]\!]\ \{x{=}v\}, \Sigma} \qquad \frac{D(X) = P \qquad v\ [\![P]\!]_D\ \Sigma}{v\ [\![X]\!]_D\ \Sigma}$$

The linearity of patterns ensures that whenever the environment combination Σ_1, Σ_2 is invoked by the rules, it is well-defined. Also, $v \; [\![P]\!] \; \Sigma$ implies that $dom(\Sigma) = fv(P)$. Note that the relation $[\![P]\!]$ is not a function thanks to the rules for patterns P_1, P_2 and $P_1 \cup P_2$, which can relate a value to more than one environment. This is different from XDUCE which imposed additional disambiguation conditions on these rules.

3.1 Lemma: If pattern P is defined under table D and D' extends D, then $[\![P]\!]_D = [\![P]\!]_{D'}$.

3.2 Lemma: For any value v, we have $v \; [\![\text{ANY}]\!]_{D_{\text{std}}} \; \{\}$.

A pattern whose set of free variables is empty is called a *type*. Types are ranged over by T and the set of all types is written *Type*. We say that the *type of the pattern* P, denoted $type(P)$, is the type obtained from P by erasing all variable binders, i.e. by changing all fragments of the form R as x to just R, recursively. If necessary, this involves traversing pattern name definitions in the defining table D and duplicating them as types, using fresh names. A *typing environment* Γ is a mapping from a finite subset $dom(\Gamma)$ of *Var* to *Type*.

If T is a type and $v \; [\![T]\!]_D \; \Sigma$, then $\Sigma = \{\}$. Thus, we can identify the relation $[\![T]\!]_D$ with the set $[\![T]\!]_D = \{ \, v \mid v \; [\![T]\!]_D \; \{\} \, \}$. This enables us to define *subtyping* on types:

$$T_1 <:_D T_2 \quad \text{iff} \quad [\![T_1]\!]_D \subseteq [\![T_2]\!]_D,$$

and on patterns:

$$P_1 <:_D P_2 \quad \text{iff} \quad type(P_1) <:_D type(P_2).$$

A type T (defined under $D \supseteq D_{\text{std}}$) is called a *tree type* if $T <:_D$ `*{`ANY`}`—motivated by the fact that all values in $[\![T]\!]_D$ are trees (i.e., singleton sequences). Similarly, a pattern P is a *tree pattern* if $type(P)$ is a tree type.

3.3 Lemma: Let R be a tree pattern defined under table D. Let v be a sequence value $v = t_1, \ldots, t_n$. Then, for any Σ

$$v \; [\![\text{ANY}, P, \text{ANY}]\!]_D \; \Sigma \quad \text{iff} \quad \exists \, i. \; t_i \; [\![P]\!]_D \; \Sigma.$$

The pattern matching relation $[\![P]\!]_D$ satisfies the following fundamental theorem (paraphrased from [8, 7]).

3.4 Theorem [Hosoya–Pierce, 2001]: For any type T and a pattern P, both defined under table D, there exist a table $D' \supseteq D$ and a typing environment Γ with $dom(\Gamma) = fv(P)$ such that, for any $x \in fv(P)$,

$$[\![\Gamma(x)]\!]_{D'} = \{ \, \Sigma(x) \mid \exists \, v \in [\![T]\!]_D. \; \exists \, \Sigma. \; v \; [\![P]\!]_D \; \Sigma \, \}.$$

We denote the situation described in the theorem by $T \; [\![P]\!]_{D \subseteq D'} \; \Gamma$ and say that the typing environment Γ represents the result of *pattern type inference* matching the input type T against pattern P.

This theorem tells us that, for any input type T, pattern P, and variable x occurring in P, the *set* of all values that could ever get bound to x as the result of matching against P all possible values from $[\![T]\!]_D$ is representable by a *type* (namely, the type $\Gamma(x)$). Indeed, there is an algorithm computing Γ and D' such that $T \; [\![P]\!]_{D \subseteq D'} \; \Gamma$ [8].

4 Paths

We now turn to paths. The following grammar defines a simplified language of paths based on the forward-axis fragment of XPATH: [2]

p	$::=$		*paths*
	$\mid$	s	single step
	$\mid$	$s\,\texttt{/}\,p$	multiple steps
s	$::=$		*steps*
	$\mid$	$a :: N$	simple step
	$\mid$	$s\,\texttt{[}q\texttt{]}$	step with predicate
N	$::=$	$L\texttt{\{}T\texttt{\}}$	node test
q	$::=$		*predicates*
	$\mid$	p	path
	$\mid$	$q\,\texttt{or}\,q$	disjunction
	$\mid$	$q\,\texttt{and}\,q$	conjunction
a	$::=$		*axes*
	$\mid$	`self`	
	$\mid$	`child`	
	$\mid$	`desc`	
	$\mid$	`dos`	

Note that node tests $N = L\texttt{\{}T\texttt{\}}$ happen to be patterns (more precisely, types) from Section 3. Because of this, we have to say sometimes that a path p is *defined under* a table D, which means that all node test patterns occurring in p are defined under D.

Some sample paths (all defined under D_{std}) that could be written in this language are

- $\texttt{dos} :: a\texttt{\{}\textsc{any}\texttt{\}}$ that returns all subtrees of the input tree whose top node is labeled with a,
- $\texttt{dos} :: \texttt{*\{()\}}$ that returns all leaves of the input tree,
- $\texttt{child} :: a\texttt{\{}\textsc{any}\texttt{\}}\,\texttt{/}\,\texttt{child} :: b\texttt{\{}\textsc{any}\texttt{\}}\,\texttt{/}\,\texttt{child} :: c\texttt{\{}\textsc{any}\texttt{\}}$ returning all c-labeled grand-grand children of the input tree that are also children of b-labeled nodes, that are in turn children of a-labeled children of the input tree,
- $\texttt{self} :: a\texttt{\{}\textsc{any}\texttt{\}[}\texttt{child} :: b\texttt{\{}\textsc{any}\texttt{\}}\,\texttt{and}\,\texttt{child} :: c\texttt{\{}\textsc{any}\texttt{\}]}\,\texttt{/}\,\texttt{child} :: c\texttt{\{}d\texttt{\{}\textsc{any}\texttt{\}\}}$ that checks if the input tree is labeled with a and has children labeled with b and c; if so, it returns only those c-labeled children that have a single, d-labeled, child.

This language mimics XPATH's unabbreviated syntax, in that axes are mentioned explicitly in every step. XPATH also offers abbreviated notation for the more popular `child` and `dos` axes. For example, an abbreviated path $a\,\texttt{/}\,b$ would correspond, in our notation, to $\texttt{child} :: a\texttt{\{}\textsc{any}\texttt{\}}\,\texttt{/}\,\texttt{child} :: b\texttt{\{}\textsc{any}\texttt{\}}$, and $a\,\texttt{//}\,b$ would correspond to $\texttt{child} :: a\texttt{\{}\textsc{any}\texttt{\}}\,\texttt{/}\,\texttt{dos} :: \texttt{*\{}\textsc{any}\texttt{\}}\,\texttt{/}\,\texttt{child} :: b\texttt{\{}\textsc{any}\texttt{\}}$. We use only the unabbreviated syntax for the sake of uniformity.

Following the structure of the syntax, the semantics of paths is given by several mutually dependent relations and predicates on trees: $[\![\cdot]\!]^{\mathsf{p}}$, $[\![\cdot]\!]^{\mathsf{s}}$, $[\![\cdot]\!]^{\mathsf{q}}$ and $[\![\cdot]\!]^{\mathsf{a}}$. As with patterns, the semantics is parameterized by a table D, but we again omit this parameter from almost all definitions in order to lighten the notaion.

The meaning of a path p is a relation $[\![p]\!]^{\mathsf{p}}_D \subseteq \mathit{Tree} \times \mathit{Tree}$ given by the following rules.

$$\frac{t\ [\![s]\!]^{\mathsf{s}}\ u}{t\ [\![s]\!]^{\mathsf{p}}\ u} \qquad\qquad \frac{t\ [\![s]\!]^{\mathsf{s}}\ t' \qquad t'\ [\![p]\!]^{\mathsf{p}}\ u}{t\ [\![s\,\texttt{/}\,p]\!]^{\mathsf{p}}\ u}$$

[2] The tokens `desc` and `dos` correspond to XPATH axis names `descendant` and `descendant-or-self`.

When $t\ [\![p]\!]^{\mathsf{p}}\ u$, we say that tree u is among the trees resulting from applying path p to tree t.

The meaning of a step s is a relation $[\![s]\!]^{\mathsf{s}}_D \subseteq \mathit{Tree} \times \mathit{Tree}$ given by the following rules.

$$\frac{t\ [\![a]\!]^{\mathsf{a}}\ u \qquad u\ [\![N]\!]_D\ \Sigma}{t\ [\![a :: N]\!]^{\mathsf{s}}_D\ u} \qquad\qquad \frac{t\ [\![s]\!]^{\mathsf{s}}\ u \qquad u \in [\![q]\!]^{\mathsf{q}}}{t\ [\![s\,\mathtt{[}q\mathtt{]}]\!]^{\mathsf{s}}\ u}$$

When $t\ [\![s]\!]^{\mathsf{s}}\ u$, we say that tree u is among the trees resulting from applying step s to tree t.

The meaning of a path predicate q is a predicate $[\![q]\!]^{\mathsf{q}}_D \subseteq \mathit{Tree}$ given by the following rules.

$$\frac{t\ [\![p]\!]^{\mathsf{p}}\ u}{t \in [\![p]\!]^{\mathsf{q}}} \qquad \frac{t \in [\![q_1]\!]^{\mathsf{q}} \qquad t \in [\![q_2]\!]^{\mathsf{q}}}{t \in [\![q_1\ \mathtt{and}\ q_2]\!]^{\mathsf{q}}} \qquad \frac{t \in [\![q_1]\!]^{\mathsf{q}}}{t \in [\![q_1\ \mathtt{or}\ q_2]\!]^{\mathsf{q}}} \qquad \frac{t \in [\![q_2]\!]^{\mathsf{q}}}{t \in [\![q_1\ \mathtt{or}\ q_2]\!]^{\mathsf{q}}}$$

When $t \in [\![q]\!]^{\mathsf{q}}$, we say that tree t matches predicate q.

The meaning of an axis a is a relation $[\![a]\!]^{\mathsf{a}} \subseteq \mathit{Tree} \times \mathit{Tree}$ given by the following rules. (Observe that $[\![a]\!]^{\mathsf{a}}$ does not need to be parameterized by D.)

$$\frac{}{t\ [\![\mathtt{self}]\!]^{\mathsf{a}}\ t} \qquad\qquad \frac{t = l\{t_1, \ldots, t_n\} \qquad 1 \le i \le n}{t\ [\![\mathtt{child}]\!]^{\mathsf{a}}\ t_i}$$

$$\frac{t = l\{t_1, \ldots, t_n\} \qquad 1 \le i \le n \qquad t_i\ [\![\mathtt{dos}]\!]^{\mathsf{a}}\ u}{t\ [\![\mathtt{desc}]\!]^{\mathsf{a}}\ u} \qquad \frac{}{t\ [\![\mathtt{dos}]\!]^{\mathsf{a}}\ t} \qquad \frac{t\ [\![\mathtt{desc}]\!]^{\mathsf{a}}\ u}{t\ [\![\mathtt{dos}]\!]^{\mathsf{a}}\ u}$$

When $t\ [\![a]\!]^{\mathsf{a}}\ u$, we say that tree u is among the trees resulting from applying axis a to tree t.

Finally, the semantics of paths can be extended to operate on sequence values, $[\![p]\!]^{\mathsf{p}}_D \subseteq \mathit{Val} \times \mathit{Tree}$, via the natural "lifting" rule

$$\frac{v = t_1, \ldots, t_n \qquad t_i\ [\![p]\!]^{\mathsf{p}}\ u}{v\ [\![p]\!]^{\mathsf{p}}\ u}$$

This formulation of paths semantics is similar to formalizations of XPATH 1.0 [16] given elsewhere, e.g. [15, 6]. (One variation is due to node tests, which are a types-based feature of XPATH 2.0 [17]. However, instead of matching against type *names* inside values, as XPATH does (the type names are added during validation against a W3C Schema), we give meaning to node tests by reusing the semantics of regular tree patterns from Section 3.) This direct approach differs from XPATH 2.0 Formal Semantics [18] that defines a core language of `for` loop expressions operating on sequences resulting from axis projections and gives meaning to path steps and predicates by translation into this core language.

Given a tree type T and a path p we write $T\ [\![p]\!]^{\mathsf{p}}$ to denote the set of trees

$$T\ [\![p]\!]^{\mathsf{p}} = \{\, u \mid \exists\, t \in [\![T]\!].\ t\ [\![p]\!]^{\mathsf{p}}\ u \,\}.$$

In the following section, we will see that this set is actually representable by a regular type.

We finish this section with a few remarks on the relationship between the language of paths introduced above and XPATH:

- We use only the *downward* subset of XPATH *forward* axes, which are the axes that can be supported in the data model of immutable values with sharing. This excludes all *reverse* axes (e.g., `parent` and `ancestor`), as well as the *forward* axes `following-sibling` and `following` whose semantics refers to the parent node of the context node and cannot be supported in our data model.
- For a similar reason, we did not include the root step, as in (abbreviated) patterns like `/` a and `//` a.
- Our semantics of node tests is structural (and, as noticed above, coincides with the semantics of tree patterns), in contrast to the XPATH 2.0 nominal semantics grounded in W3C Schema validation.
- Predicates are restricted to conjunctions and disjunctions of paths—in contrast to XPATH, where they can be arbitrary boolean-valued computations involving arithmetic, function calls, etc.

- We have refrained from using negation in path predicates. It would have to be translated (in Section 5) into negation on patterns, which would require to modify the semantics of patterns to explicitly model matching failure (e.g., as a special $\perp$ environment). We felt this complication would be out of balance with the ideas we present here.

- This semantics does not support positional predicates since they, compared to path predicates (as well as to other path features of our fragment), have to be defined in terms of intermediate sequences. Consider, for example, the path $\texttt{child}::a\texttt{[3]}$ with the positional predicate `[3]` when applied to a tree of type $\texttt{*\{}a\texttt{\{\}} \cup b\texttt{\{\}\}}$. This path selects all a-children (ignoring all possible b-children in between), and picks the third among them. In a sense, the a-children are materialized into a new sequence—a one that does not exist as a contiguous fragment of the input tree. In contrast, our semantics can only analyze the value structure that is already present in the input. The latter property is common with the semantics of patterns and is essential for the success of the translation in the next section.

5 Paths Into Patterns

The translation from paths into regular patterns is given by a collection of mutually dependent functions for translating a path p, a step s, a predicate q, or an axis a into a pattern. Each of the functions additionally takes as an argument and returns as a result a pattern definition table: the output table D' is possibly an extension of the input table D with new pattern definitions that have been generated during the translation of axes. Also, all functions (except the predicate translation) take a third argument, a pattern R, which may be thought of as a "continuation" carrying the result of an earlier translation.

The translation of a path p into a pattern P (with continuation R, taking definitions D and yielding definitions D'), written $D\,;p \xrightarrow{R}_{\mathsf{p}} P\,;D'$, is defined by the following rules:

$$\frac{D\,;s \xrightarrow{R}_{\mathsf{s}} P\,;D'}{D\,;s \xrightarrow{R}_{\mathsf{p}} P\,;D'} \qquad \frac{D\,;p \xrightarrow{R}_{\mathsf{p}} R'\,;D' \qquad D\,;s \xrightarrow{R'}_{\mathsf{s}} P\,;D''}{D\,;s\,\texttt{/}\,p \xrightarrow{R}_{\mathsf{p}} P\,;D''}$$

The translation of a step s into a pattern P, written $D\,;s \xrightarrow{R}_{\mathsf{s}} P\,;D'$, is defined by the following rules:

$$\frac{D\,;a \xrightarrow{N\cap R}_{\mathsf{a}} P\,;D'}{D\,;a::N \xrightarrow{R}_{\mathsf{s}} P\,;D'} \qquad \frac{D\,;q \rightarrow_{\mathsf{q}} R'\,;D' \qquad D\,;s \xrightarrow{R'\cap R}_{\mathsf{s}} P\,;D''}{D\,;s\,\texttt{[}q\texttt{]} \xrightarrow{R}_{\mathsf{s}} P\,;D''}$$

The translation of a predicate q into a pattern P, written $D\,;q \rightarrow_{\mathsf{q}} P\,;D'$, is defined by the following rules:

$$\frac{D\,;q \xrightarrow{\text{ANY}}_{\mathsf{p}} P\,;D'}{D\,;q \rightarrow_{\mathsf{q}} P\,;D'} \qquad \frac{D\,;q_1 \rightarrow_{\mathsf{q}} P_1\,;D' \qquad D'\,;q_2 \rightarrow_{\mathsf{q}} P_2\,;D''}{D\,;q_1\,\texttt{and}\,q_2 \rightarrow_{\mathsf{q}} P_1 \cap P_2\,;D''} \qquad \frac{D\,;q_1 \rightarrow_{\mathsf{q}} P_1\,;D' \qquad D'\,;q_2 \rightarrow_{\mathsf{q}} P_2\,;D''}{D\,;q_1\,\texttt{or}\,q_2 \rightarrow_{\mathsf{q}} P_1 \cup P_2\,;D''}$$

The translation of an axis a into a pattern P, written $D\,;a \xrightarrow{R}_{\mathsf{a}} P\,;D'$, is defined by the following rules (DESC and DOS in the final two rules are fresh pattern names):

$$\frac{}{D\,;\texttt{self} \xrightarrow{R}_{\mathsf{a}} R\,;D} \qquad \frac{}{D\,;\texttt{child} \xrightarrow{R}_{\mathsf{a}} \texttt{*\{}\text{ANY}, R, \text{ANY}\texttt{\}}\,;D}$$

$$\frac{\begin{array}{c}\text{DESC} \notin dom(D)\\ D' = D \cup \{\text{DESC} = \texttt{*\{}\text{ANY}, (R \cup \text{DESC}), \text{ANY}\texttt{\}}\}\end{array}}{D\,;\texttt{desc} \xrightarrow{R}_{\mathsf{a}} \text{DESC}\,;D'} \qquad \frac{\begin{array}{c}\text{DOS} \notin dom(D)\\ D' = D \cup \{\text{DOS} = R \cup \texttt{*\{}\text{ANY}, \text{DOS}, \text{ANY}\texttt{\}}\}\end{array}}{D\,;\texttt{dos} \xrightarrow{R}_{\mathsf{a}} \text{DOS}\,;D'}$$

Given a path p, translating it to a pattern P amounts to picking a variable x (which is going to bind, in P, to results of p) and invoking the translation $D\,;p \xrightarrow{\text{ANY}\ \texttt{as}\ x}_{\mathsf{p}} P\,;D'$, where D is a table under which p is defined. (Here, as well is in the rule for translating the elementary path predicate, one could prefer to use

ANYONE rather than ANY as the "seed pattern", but this is not necessary, since this pattern eventually gets intersected with a node test pattern.)

This translation captures precisely the semantics of paths in the following sense: the results of applying p to a value v are exactly the values that are bound to x in all possible environments resulting from matching v against P. This result (Theorem 5.4 below) depends on Lemmas 5.2 and 5.3, which relate the above four translations to the corresponding four semantics of paths, steps, predicates, and axes.

But first, we need a few preliminary lemmas.

5.1 Lemma: If $D\,;q \rightarrow_{\mathsf{q}} P\,;D'$, then $fv(P) = \emptyset$. Consequently, $v\ [\![P]\!]_{D'}\ \Sigma$ implies $\Sigma = \{\}$.

5.2 Lemma: Let R be a tree pattern defined under table D. Then, for any axis a,

$$D\,;a \xrightarrow{R}_{\mathsf{a}} P\,;D' \quad \text{implies} \quad [\![a]\!]^{\mathsf{a}} \circ [\![R]\!]_D = [\![P]\!]_{D'}.$$

5.3 Lemma: Let pattern R be defined under table D. Then for any path p, any step s, and any predicate q, all of the following are true:

$$\begin{array}{lll} D\,;p \xrightarrow{R}_{\mathsf{p}} P\,;D' & \text{implies} & [\![p]\!]^{\mathsf{p}}_D \circ [\![R]\!]_D = [\![P]\!]_{D'}, \\ D\,;s \xrightarrow{R}_{\mathsf{s}} P\,;D' & \text{implies} & [\![s]\!]^{\mathsf{s}}_D \circ [\![R]\!]_D = [\![P]\!]_{D'}, \\ D\,;q \rightarrow_{\mathsf{q}} P\,;D' & \text{implies} & [\![q]\!]^{\mathsf{q}}_D = \{\, t \mid \quad t\ [\![P]\!]_{D'}\ \{\}\,\}. \end{array}$$

The translation correctness theorem now follows from the first clause of Lemma 5.3.

5.4 Theorem: Let p be a path defined under table D, and x be a variable. If $D\,;p \xrightarrow{\text{ANY as } x}_{\mathsf{p}} P\,;D'$, then, for any value v,

$$\{\, u \mid v\ [\![p]\!]^{\mathsf{p}}_D\ u\,\} = \{\, \Sigma(x) \mid v\ [\![P]\!]_{D'}\ \Sigma\,\}.$$

5.5 Corollary: For any type T and any path p the set $T\ [\![p]\!]^{\mathsf{p}}$, the results of applying p to values from T, is representable as a regular type.

Proof: Suppose both T and p are defined under the table D. By a simple calculation involving Theorems 5.4 and 3.4 we can obtain that there exists a typing environment Γ such that $T\ [\![p]\!]^{\mathsf{p}}_D\ = [\![\Gamma(x)]\!]_D$, which means that the set $T\ [\![p]\!]^{\mathsf{p}}_D$ is represented by the regular type $\Gamma(x)$. □

This corollary states that the translation from paths to patterns proposed here provides not only a mechanism for evaluating paths, but also, when combined with the pattern type inference algorithm from [8], it provides a type system for the XPATH fragment which is *precise*, i.e. sound and complete w.r.t. the evaluation semantics of paths. This is in contrast to the typing rules in XQuery 1.0 and XPath 2.0 Formal Semantics [18], which are sound, but not complete (with respect to the same fragment of XPATH that we consider here).

6 Related Work

We are not aware of other work formally comparing functionality of paths and regular patterns. Therefore, this section briefly mentions several works broadly related to the subject of the paper.

In most language designs based on regular patterns ambiguity was addressed by proposing a disambiguation policy for selecting a single answer among multiple possibilities. The original paper on XDUCE regular patterns [8] assigned numeric *choice sequences* (recording choices made by union patterns) to each successful match, and the match with the smallest choice sequence in the dictionary order was taken as the result. A different implementation of the same disambiguation policy is described by Frisch and Cardelli [3]. Vansummeren[13] studies two other disambiguation policies.

Another approach to returning multiple bindings from a pattern that is worth a closer look is using non-linear patterns, i.e. patterns that bind a variable more than once during a single parse of the input value (as contrasted to ambiguous patterns where multiple bindings result from alternative parses of the same value). CDUCE [1] and regular patterns proposal[2] for Haskell are a couple language designs that exploit non-linearity in recursive patterns to arrange iteration over sequences.

Murata [11] studies a language of path expressions extended with conditions on siblings of nodes in a path that are similar to regular patterns without variables.

7 Future Work

The multi-match semantics of patterns and paths presented in this paper are "relational", in the sense that they relate an input value to a *single* output value or environment (of which there can be many, though). This can be contrasted to an approach when the input value is mapped, functionally, to a list representing the totality of all answers. Even though the relational approach helped us to obtain results presented here (after a few false starts with the other approach), the current semantics does not capture the aspect of XPath related to "provenance" of results. More concretely, if a path selects several distinct, but structurally equivalent, subtrees from the input value, our semantics cannot determine how many are there. This can be rectified, however, by a straightforward modification of the semantics to relate an input value to locations in the input value rather than to output values corresponding to those locations.

There exists literature (e.g., [12]) on translating XPath with reverse axes into XPath with only forward axes, based on the assumption that the path possibly containing reverse axes is always applied to the document root. Combining such a translation with the approach presented here could translate paths involving all of XPath axes into regular patterns. The most value of reverse axes, however, comes from use cases where the document root is not explicitly given, or, in other words, where a path's reverse fragments traverse the input tree upwards beyond its top node. So, there does not appear to be a significant pragmatic advantage from combining the two translations.

An interesting question is translating the other way around, from patterns to paths. For example, can every ambiguous pattern binding exactly one variable be translated into an equivalent path? Is there a natural point of view on paths that can ascribe them a semantics of multiple variable binders, to parallel the ambiguous view of patterns that allowed us to ascribe them a multiple-result semantics?

Last, but not least, is implementing the proposal presented here in our XML processing language Xtatic [5, 4]. While doing so, we hope to design a pattern language that mixes together paths and patterns.

Acknowledgments

We are grateful to Michael Levin and Alan Schmitt who contributed to many of our discussions on the subject of this paper.

References

[1] V. Benzaken, G. Castagna, and A. Frisch. CDuce: An XML-centric general-purpose language. In *ACM SIGPLAN International Conference on Functional Programming (ICFP), Uppsala, Sweden*, pages 51–63, 2003.

[2] N. Broberg, A. Farre, and J. Svenningsson. Regular expression patterns. In *ACM SIGPLAN International Conference on Functional Programming (ICFP), Snowbird, Utah*, 2004.

[3] A. Frisch and L. Cardelli. Greedy regular expression matching. In *ICALP*, Jan. 2004.

[4] V. Gapeyev, M. Y. Levin, B. C. Pierce, and A. Schmitt. XML goes native: Run-time representations for Xtatic. Technical Report MS-CIS-04-23, University of Pennsylvania, Oct. 2004.

[5] V. Gapeyev and B. C. Pierce. Regular object types. In *European Conference on Object-Oriented Programming (ECOOP), Darmstadt, Germany*, 2003. A preliminary version was presented at FOOL '03.

[6] G. Gottlob, C. Koch, and R. Pichler. XPath processing in a nutshell. *ACM SIGMOD Record*, 32(1):12–19, 2003.

[7] H. Hosoya. Regular expression pattern matching—a simpler design. Technical Report 1397, RIMS, Kyoto University, 2003.

[8] H. Hosoya and B. C. Pierce. Regular expression pattern matching. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), London, England*, 2001. Full version in *Journal of Functional Programming*, 13(6), Nov. 2003, pp. 961–1004.

[9] H. Hosoya, J. Vouillon, and B. C. Pierce. Regular expression types for XML. In *Proceedings of the International Conference on Functional Programming (ICFP)*, 2000.

[10] K. Z. M. Lu and M. Sulzmann. XHaskell: Regular expression types for Haskell. Technical Report TRC9/04, National University of Singapore, 2004.

[11] M. Murata. Extended path expressions for XML. In *PODS*, pages 126–137, 2001.

[12] D. Olteanu, H. Meuss, T. Furche, and F. Bry. XPath: Looking forward. In *EDBT Workshop on XML Data Management (XMLDM)*, volume 2490 of *Lecture Notes in Computer Science*, pages 109–127. Springer, 2002.

[13] S. Vansummeren. Unique pattern matching in strings, 2003. http://arxiv.org/abs/cs.PL/0302004.

[14] W3C. XSL Transformations (XSLT), 1999. `http://www.w3.org/TR/xslt`.

[15] P. Wadler. Two semantics for XPath, 2000. A note, available at http://homepages.inf.ed.ac.uk/wadler/.

[16] XML Path Language (XPath) Version 1.0, W3C Recommendation, Nov. 1999. http://www.w3c.org/TR/xpath.

[17] XML Path Language (XPath) 2.0, W3C Working Draft, July 2004. http://www.w3.org/TR/xpath20/.

[18] XQuery 1.0 and XPath 2.0 Formal Semantics, W3C Working Draft, Feb. 2004. http://www.w3c.org/TR/query-semantics/.

[19] XQuery 1.0: An XML Query Language, W3C Working Draft, July 2004. http://www.w3.org/TR/xquery/.