Regular Expression Pattern Matching — A Simpler Design —

Haruo HOSOYA
RIMS, Kyoto University
hahosoya@kurims.kyoto-u.ac.jp

February 6, 2003

Abstract

Regular expression pattern matching is a programming language feature designed for the analysis and decomposition of XML documents. Since the basis is regular expressions, a pattern may ambiguously match an input value and thus yield different bindings from this. Our previous design has dealt with this by a first-match semantics, where we have imposed certain priority rules for patterns and taken the first choice according to these rules. While this semantics is convenient in some cases, it behaves in a quite unintuitive way in other cases. In this paper, we propose an alternative design of regular expression pattern matching. In this, we drop the priority rules but add an ambiguity check on patterns to ensure them to always yield unique bindings. We give two algorithms related to this design of pattern matching: type inference for pattern variables and ambiguity check. The type inference is similar to our previously developed technique in that it computes "exact" types for variables from local type information, but it improves the previous one in two respects, adaptation to the new semantics and handling of pattern variables in non-tail positions. The ambiguity check is a simple algorithm based on product construction of tree automata, but solves a subtle problem arising from silent transitions of automata.

1 Introduction

Regular expression pattern matching [HP02] is a programming feature that has emerged from the design of the statically typed language XDuce for XML processing [HP03]. This feature, being derived from pattern matching facilities commonly found in functional languages [LVD+96, AM91, PHH+93, etc], allows us to analyze and decompose tree structure of XML documents. Since the underlying structural descriptions, i.e., types, of documents are based on regular expressions, patterns here are, as a natural choice, also based on regular expressions.

One issue in designing this style of pattern matching is ambiguity. That is, we may write a pattern that can match an input value in several ways, and how should the language treat this? For example, we can write the following pattern match

```
match v with
  Email*, (Email as x), Email*  ->  ... (* do something with x *)
```

for extracting one node of type `Email` (which is defined somewhere else) from a sequence `v` of nodes each of type `Email` and binding the variable `x` to the extracted node. In our previous proposal, we have adopted a first-match semantics, where we impose certain priority rules on patterns and take the "first" choice among several possibilities according to these rules.

In this paper, we propose an alternative design of regular expression pattern matching. In this, we drop the priority rules on patterns but instead add a static ambiguity check so as to ensure patterns to always match input values uniquely. We have found that this design is simpler than the previous and therefore preferable for users. Specifically, the definition of the first-match semantics is rather complicated and artificial in some places, and this raises some patterns whose behaviors are against our "normal" intuition. By requiring unambiguity, the meanings of such patterns become clearer. Although unambiguous patterns can be larger than ambiguous ones, this increase seems to be usually acceptable. (We show examples in Section 2.)

For our new design of pattern matching, we have developed two algorithms needed to integrate it in a statically typed language, namely, type inference for pattern variables and ambiguity check.

The type inference, intended to be used as a part of an underlying typechecker, supplies suitable types for variables appearing in patterns from type information given by the typechecker. The inference is both *local* and *locally precise*. By local, we mean that we compute types for variables only from the pattern itself and the type for input values, the latter being given from the underlying typechecker. (We do not take into account distant information such as from pattern bodies.) By locally precise, we mean that the type computed for a variable contains *exactly* the set of values that may be bound to the variable, with the conservative assumption that *all* the values from the input type may be passed to the pattern. (Note that, at actual run-time, not all values may be passed to the pattern.)

In the previous paper [HP02], we have already presented a type inference algorithm whose specification is exactly the same as above. The algorithm in this paper is different in two ways. First, it infers types based on the different pattern semantics. That is, since we do not have the first-match priority rules any more, the inference algorithm does not need to take into account "the values *not* matched by the patterns with higher priority." This greatly simplifies the algorithm and, in particular, reduces its complexity from exponential to quadratic. The second difference is that the new algorithm eliminates the previous technical limitation that allows the precise inference only for variables

in "tail-positions." We achieve this by adopting a new automata encoding that involves "sequence-accumulators" (Section 4).

Our ambiguity checking algorithm is based on product construction of automata. This is not surprising by itself—it could be found as an exercise of some undergraduate textbook. However, in our case, (1) we consider tree automata (rather than string automata), (2) they contain silent transitions, and (3) we treat ambiguity for a restricted set of input values described by a given type (rather than for all input values). The combination of all of them turns out to require a quite delicate treatment. Although the ambiguity checking problem for regular expressions and automata has been a classic problem and a lot of related discussions can be found in the literature (e.g., [BEGO71, BK93]), we have not found any algorithm similar to ours. The closest related work is Kawaguchi's ambiguity checking algorithm for tree regular expressions [Kaw01]. We will discuss this in details in Section 7.

In the next section, we give some examples to introduce basic concepts and motivate the new design of pattern matching. Section 3 presents the definitions of values, types, and patterns of the surface language as well as the inference and ambiguity problems. In Section 4, we define tree automata with variables, which will be used for describing the type inference (Section 5) and the ambiguity checking algorithms (Section 6). Section 7 discusses the relationship with other work and Section 8 concludes the paper.

2 Examples

In our language, values are sequences of labeled values or base values and thus represent fragments of XML documents. Types describe such sequences by using regular expressions. For example, consider the following type definitions (which give names to types).

```
type Person = person[Name,Email*,Tel]
type Name   = name[String]
type Email  = email[String]
type Tel    = tel[String]
```

Type constructors of the form `label[...]` classify tree nodes with the label `label` (i.e., XML structures of the form `<label>...</label>`). Thus, the inhabitants of the types `Name`, `Email`, and `Tel` are all strings with an appropriate identifying label. Type constructors of the form `T*` denote a sequence of arbitrarily many `T`s. Thus, the inhabitants of the type `Person` are nodes labeled `person` whose content is a sequence consisting of a name, zero or more email addresses, and a telephone number.

Regular expression pattern matches analyze the structure of an input value and extract substructures of them. Syntactically, patterns are exactly the same as types except that we can insert "variable bindings" in them. For example, we can write the following pattern match for extracting the `tel` node from a given `person`.

```
match person with
  person[Name, Email*, (Tel as x)]
    ->  ... (* do something with x *)
```

This matches any values of type `Person` and binds `x` to the `tel` node. This pattern happens to have the structure as the input type, but it is not necessary in general. For example, we can replace the pattern `Name,Email*` in the above with the `Any` pattern, which matches any sequences.

```
match person with
  person[Any, (Tel as x)]  ->  ...
```

Using `Any` is convenient since it allows us to just ignore the parts that we do not care about.

However, this idiom sometimes causes patterns to behave rather strangely in the first-match semantics. Let us consider a slight variation. Suppose that the type `Person` is actually defined as follows

```
type Person = person[Name,Email*,Tel?]
```

where the `Tel` part is now optional. Accordingly, the pattern for extracting this part would become:

```
match person with
  person[Any, (Tel? as x)]  ->  ...
```

However, in the first-match semantics, this pattern always binds `x` to the empty sequence. Understanding why is slightly complicated. First, the `Any` pattern tries to capture the longest possible sequence. This is because `Any` is actually recursively defined as

```
type Any = ~[Any],Any | ()
```

(where `~` matches any label and `()` is the empty sequence type) and the first choice is taken as often as possible. Likewise, the `Tel?` pattern also tries to capture the longest possible sequence since it is actually an abbreviation of

```
Tel | ()
```

and the first choice is taken first. Finally, when these two patterns are concatenated, the first one is given higher priority. Consequently, the `Any` pattern always captures the whole sequence, leaving the `Tel?` pattern only the empty sequence.

In our new design, when a given pattern can "parse" an input value in multiple ways, then the system blames it as ambiguous. For example, the pattern in the last example is ambiguous since any `person`'s content that ends with a `tel` can be parsed in two ways. To make it unambiguous, we need to be more explicit, e.g.,

```
match person with
  person[^tel[Any]*, (Tel? as x)]  ->  ...
```

(where `^tel` matches any label except `tel`). The pattern is now slightly more verbose, but its meaning is clearer.

Our notion of ambiguity takes the input type into account. That is, we check if the given pattern is ambiguous for some value from the input type. Ambiguity for non-input values does not matter. For example, suppose that we have the following pattern where the input type is `Email*,Tel,Addr?`:

```
match v with
  Any, (Tel as x), Any  ->  ...
```

This pattern is unambiguous since any input value has only one `tel` and therefore there is only one parse with this pattern. Note that it *is* ambiguous for non-input values such as those with two `tel`s. If patterns were required to be ambiguous for all inputs, then the patterns would become unnecessarily bigger. Therefore considering input types helps patterns more concise, and, further, it seems in our experiences that the increase of the size of a pattern by disambiguation is relatively acceptable because of this feature.

In a concrete language design, there are actually two possible decisions for what the system should do when it detects ambiguity: rejection or warning. Usually, an ambiguous pattern signals a programming error. However, we have found that, in some cases, writing ambiguous patterns is reasonable. One typical case is when the application program knows, from implicit assumptions that cannot be expressed in types, that there is only one possible parse. For example, suppose that we have a simple book database of type `Book*` where

```
type Book = book[key[String], title[String], author[String]]
```

where we assume that there is only one `book` entry with the same `key` field value. We can extract a book with a specified key from this database by writing the following pattern match.

```
match db with
  Book*,
  book[key["Pierce2002"],
       title[String as t],
       author[String as a]],
  Book*  ->  ...
```

Note that the above assumption for keys guarantees that the entry yielded by this pattern match is unique.

Since writing a unambiguous pattern is sometimes more complicated than an ambiguous one, requiring disambiguation even in situations that do not necessitate it can be a heavy burden for the user. (In the above pattern, we would only have to replace the first occurrence of `Book` with a type representing `book`s with keys other than `"Pierce2002"`. However, this could become more cumbersome if keys have a complex structure.) Therefore, in the XDuce language, we decided to yield a warning for ambiguity rather than an error. In the case that the user writes an ambiguous pattern and ignores the warning, the semantics of pattern matching is to choose an arbitrary parse among multiple possibilities ("nondeterministic semantics").

3 Surface Language Definitions

In this section, we give formal definitions for values, types, and patterns and then specify type inference and ambiguity check in terms of them.

3.1 Values

We assume a (possibly infinite) set L of *labels*, ranged over by `l`. A *value* `v` is a sequence of labeled values, where a labeled value is a pair of a label and a value. We use the following syntax for writing values.

`v`	::=	`()`	empty sequence
		`l[v]`	labeled value
		`v,v`	concatenation

For brevity, we omit base values such as strings from the formalization. (The changes required to add them are straightforward.)

Throughout this paper, we will use metavariables in italic and typewriter fonts interchangeably.

3.2 Types and patterns

We first define *label classes* by the following syntax:

`L`	::=	`l`	specific label
		`~`	wildcard label
		`L|L`	union
		`L\L`	difference

We write a negation `^L` as an abbreviation for `~\L`. The semantics of label classes is defined by a denotation function $[\![\cdot]\!]$ mapping label classes to sets of labels.

$$
\begin{array}{lcl}
[\![\texttt{l}]\!] & = & \{\texttt{l}\} \\
[\![\texttt{\~{}}]\!] & = & L \\
[\![\texttt{L}_1\texttt{|L}_2]\!] & = & [\![\texttt{L}_1]\!] \cup [\![\texttt{L}_2]\!] \\
[\![\texttt{L}_1\texttt{\textbackslash L}_2]\!] & = & [\![\texttt{L}_1]\!] \setminus [\![\texttt{L}_2]\!]
\end{array}
$$

We write $\texttt{l} \in \texttt{L}$ for $\texttt{l} \in [\![\texttt{L}]\!]$.

We next define the syntax and semantics of types and patterns. In our previous papers [HP03, HP02], we have defined types and patterns as separate syntactic categories. In this paper, for conciseness, we first define patterns and then types as patterns containing no variable binder.

We assume a countably infinite set of pattern names, ranged over by `X`, and a countably infinite set of variables, ranged over by `x`. Pattern expressions (or

just *patterns*) are now defined as follows.

`P`	::=	`P as x`	variable binder
		`L[X]`	label
		`()`	empty sequence
		`P,P`	concatenation
		`P|P`	choice
		`P*`	repetition

The bindings of pattern names to patterns are given by a fixed, global, mutually recursive set E of pattern definitions of the following form.

```
pat X = P
```

Note that, in the formalization, we forbid nesting of labels and require type names to occur only inside labels. This is for simplicity and does not lose generality: any definitions without these restrictions can be translated to ones with the restrictions. (To ensure that patterns correspond to *regular* tree automata, rather than context-free grammars, we would need to impose a syntactic restriction that disallows recursion "at the top level" of definitions. See [HP03] for details.) The additional regular expression operators `?` and `+` are obtained as syntactic sugar:

`P?`	$\equiv$	`P|()`
`P+`	$\equiv$	`P,P*`

Also, we assume the following fixed pattern definition in E.

```
pat Any = ~[Any]*
```

Note that we take here `*` as a primitive operator. This is unlike our previous treatment, where we defined `P*` to be a fresh name `X` where the following definition is present.

```
pat X = P,X | ()
```

The reason for throwing this style away is that (1) the syntactic restriction needed for ensuring regularity (recursion variables can occur only at the right most positions) is more complicated and (2) its only usefulness that we have identified is a longest-match behavior of patterns, which we discard anyway. (Yet another complication of our previous first-match semantics is that some patterns do not actually have the first match. See Appendix B.)

In order to ensure patterns to always yield sensible bindings, we impose a "linearity" restriction on them. Let **reachable**(`P`) be the set of all variables reachable from `P`—that is, the smallest set satisfying the following:

$$\textbf{reachable}(\texttt{P}) \quad = \quad \textbf{bv}(\texttt{P}) \ \cup \bigcup_{\texttt{X} \in \textbf{fn}(\texttt{P})} \textbf{reachable}(F(\texttt{X})),$$

where **bv**(`P`) is the set of variables bound in `P` and **fn**(`P`) is the set of pattern names appearing in `P`. We say that a pattern `P` is *linear* iff, for any (reachable) subphrase `P'` of `P`, the following conditions hold.

- $\mathtt{x} \notin \mathbf{reachable}(\mathtt{P}_1)$ if $\mathtt{P}' = \mathtt{P}_1\ \mathtt{as}\ \mathtt{x}$.
- $\mathbf{reachable}(\mathtt{P}_1) \cap \mathbf{reachable}(\mathtt{P}_2) = \emptyset$ if $\mathtt{P}' = \mathtt{P}_1\mathtt{,}\mathtt{P}_2$.
- $\mathbf{reachable}(\mathtt{P}_1) = \mathbf{reachable}(\mathtt{P}_2)$ if $\mathtt{P}' = \mathtt{P}_1\mathtt{|}\mathtt{P}_2$.
- $\mathbf{reachable}(\mathtt{P}_1) = \emptyset$ if $\mathtt{P}' = \mathtt{P}_1\mathtt{*}$.

In the following, we assume that all patterns are linear.

We define the semantics of patterns by the relation $\mathtt{v} \in \mathtt{P} \Rightarrow \mathtt{V}$, read "v is matched by P, yielding V," where an environment V is a finite mapping from variables to values (written $\mathtt{x}_1\mathtt{:}\mathtt{v}_1\mathtt{,}\ldots\mathtt{,}\mathtt{x}_n\mathtt{:}\mathtt{v}_n$). The concatenation of environments binding distinct variables is written with a comma.

$$\frac{\mathtt{v} \in \mathtt{P} \Rightarrow \mathtt{V}}{\mathtt{v} \in \mathtt{(P\ as\ x)} \Rightarrow \mathtt{x:v,V}} \qquad \text{(EP-As)}$$

$$\mathtt{()} \in \mathtt{()} \Rightarrow \emptyset \qquad \text{(EP-Emp)}$$

$$\frac{E(\mathtt{X}) = \mathtt{P} \qquad \mathtt{l} \in \mathtt{L} \qquad \mathtt{v} \in \mathtt{P} \Rightarrow \mathtt{V}}{\mathtt{l[v]} \in \mathtt{L[X]} \Rightarrow \mathtt{V}} \qquad \text{(EP-Lab)}$$

$$\frac{\mathtt{v}_1 \in \mathtt{P}_1 \Rightarrow \mathtt{V}_1 \qquad \mathtt{v}_2 \in \mathtt{P}_2 \Rightarrow \mathtt{V}_2}{\mathtt{v}_1\mathtt{,}\mathtt{v}_2 \in \mathtt{P}_1\mathtt{,}\mathtt{P}_2 \Rightarrow \mathtt{V}_1\mathtt{,}\mathtt{V}_2} \qquad \text{(EP-Cat)}$$

$$\frac{\mathtt{v} \in \mathtt{P}_1 \Rightarrow \mathtt{V}}{\mathtt{v} \in \mathtt{P}_1\mathtt{|}\mathtt{P}_2 \Rightarrow \mathtt{V}} \qquad \text{(EP-Or1)}$$

$$\frac{\mathtt{v} \in \mathtt{P}_2 \Rightarrow \mathtt{V}}{\mathtt{v} \in \mathtt{P}_1\mathtt{|}\mathtt{P}_2 \Rightarrow \mathtt{V}} \qquad \text{(EP-Or2)}$$

$$\frac{\mathtt{v}_i \in \mathtt{P} \Rightarrow \mathtt{V}_i \ \text{ for each } i}{\mathtt{v}_1, \ldots, \mathtt{v}_n \in \mathtt{P*} \Rightarrow \mathtt{V}_1, \ldots, \mathtt{V}_n} \qquad \text{(EP-Rep)}$$

Note that linearity ensures that environments that are concatenated in the conclusions of rules EP-As, EP-Cat, and EP-Rep have different domains (e.g., $\mathtt{x} \notin \mathbf{dom}(\mathtt{V})$ always holds in rule EP-As). We write $\mathtt{v} \in \mathtt{P}$ when $\mathtt{v} \in \mathtt{P} \Rightarrow \mathtt{V}$ for some V.

Types are patterns from which no variables are reachable, i.e., $\mathbf{reachable}(\mathtt{P}) = \emptyset$. Types are ranged over by S, T, U, etc.

3.3 Type inference

The goal of pattern type inference is to compute the "range" of a pattern, defined as follows. A mapping Γ from variables to types describes the range of a pattern P with respect to type T iff, for all x and v, we have

> $\mathtt{v} \in \Gamma(\mathtt{x})$ iff there exists a value $\mathtt{u} \in \mathtt{T}$ such that $\mathtt{u} \in \mathtt{P} \Rightarrow \mathtt{V}$ for some V with $\mathtt{V}(\mathtt{x}) = \mathtt{v}$.

3.4 Ambiguity

We define ambiguity in terms of the relation $\mathtt{v} \in^a \mathtt{P}$, read "v matches P ambiguously." The relation is defined by the following set of rules.

$$\frac{\mathtt{v} \in^a \mathtt{P}}{\mathtt{v} \in^a \mathtt{(P\ as\ x)}} \qquad \text{(AMB-As)}$$

$$\frac{E(\mathtt{X}) = \mathtt{P} \qquad \mathtt{l} \in \mathtt{L} \qquad \mathtt{v} \in^a \mathtt{P}}{\mathtt{l[v]} \in^a \mathtt{L[X]}} \qquad \text{(AMB-Lab)}$$

$$\frac{\mathtt{v} = \mathtt{v}_1\mathtt{,v}_2 = \mathtt{v}_1\mathtt{'},\mathtt{v}_2\mathtt{'} \qquad \mathtt{v}_1 \neq \mathtt{v}_1\mathtt{'} \qquad \mathtt{v}_1 \in \mathtt{P}_1 \qquad \mathtt{v}_2 \in \mathtt{P}_2 \qquad \mathtt{v}_1\mathtt{'} \in \mathtt{P}_1 \qquad \mathtt{v}_2\mathtt{'} \in \mathtt{P}_2}{\mathtt{v} \in^a \mathtt{P}_1\mathtt{,P}_2} \qquad \text{(AMB-Cat1)}$$

$$\frac{\mathtt{v}_1 \in^a \mathtt{P}_1 \qquad \mathtt{v}_2 \in \mathtt{P}_2}{\mathtt{v}_1\mathtt{,v}_2 \in^a \mathtt{P}_1\mathtt{,P}_2} \qquad \text{(AMB-Cat2)}$$

$$\frac{\mathtt{v}_1 \in \mathtt{P}_1 \qquad \mathtt{v}_2 \in^a \mathtt{P}_2}{\mathtt{v}_1\mathtt{,v}_2 \in^a \mathtt{P}_1\mathtt{,P}_2} \qquad \text{(AMB-Cat3)}$$

$$\frac{\mathtt{v} \in \mathtt{P}_1 \qquad \mathtt{v} \in \mathtt{P}_2}{\mathtt{v} \in^a \mathtt{P}_1\mathtt{|P}_2} \qquad \text{(AMB-Or1)}$$

$$\frac{\mathtt{v} \in^a \mathtt{P}_1}{\mathtt{v} \in^a \mathtt{P}_1\mathtt{|P}_2} \qquad \text{(AMB-Or2)}$$

$$\frac{\mathtt{v} \in^a \mathtt{P}_2}{\mathtt{v} \in^a \mathtt{P}_1\mathtt{|P}_2} \qquad \text{(AMB-Or3)}$$

$$\frac{\mathtt{v} = \mathtt{v}_1\mathtt{,..,v}_n \qquad \mathtt{v} = \mathtt{v}_1\mathtt{',..,v}_m\mathtt{'} \qquad (\mathtt{v}_1, \ldots, \mathtt{v}_n) \neq (\mathtt{v}_1\mathtt{'}, \ldots, \mathtt{v}_m\mathtt{'}) \qquad \mathtt{v}_i \in \mathtt{P} \text{ for each } i \qquad \mathtt{v}_i\mathtt{'} \in \mathtt{P} \text{ for each } i}{\mathtt{v} \in^a \mathtt{P*}} \qquad \text{(AMB-Rep1)}$$

$$\frac{\mathtt{v}_i \in \mathtt{P} \text{ for each } i \qquad \mathtt{v}_j \in^a \mathtt{P} \text{ for some } j}{\mathtt{v}_1\mathtt{,..,v}_n \in^a \mathtt{P*}} \qquad \text{(AMB-Rep2)}$$

Intuitively, an ambiguity relation is derived in exactly the same way as the corresponding matching relation, except that the former finds a place in the derivation tree where there are more than one way of matching. Such a place can be either in a concatenation pattern (captured by AMB-Cat1), an alternation pattern (captured by AMB-Or1), or a repetition pattern (captured by AMB-Rep1). In the first two cases, the given sequence can be divided in two different ways; in the third case, the given value can be matched by both of the choices.

4 Tree Automata with Variables

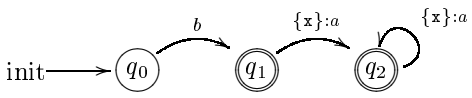
This section introduces tree automata with a capability of variable binding. We use this notion in later sections for constructing algorithms for ambiguity check and type inference for pattern variables.

4.1 Syntax and Semantics

Our tree automata are different from usual ones [CDG+99] in that each transition is associated with (1) a label class (instead of a single label) and (2) a set of variables. While the first difference is straightforward, the second needs some explanation. Consider first the following pattern.

```
b[], a[]* as x
```

We encode this pattern by the following automaton.



(We ignore the contents of labels for the moment; we'll treat them in the formulation below.) While this automaton runs, it keeps track of a "sequence accumulator" associated to `x`. When the automaton follows each transition with a, it appends the label a to the accumulator. In general, if the automaton contains multiple variables, it has multiple accumulators corresponding to them. We associate each transition with a set of variables (rather than a single variable) since variable binders can be nested, e.g., `(a[]* as x, b[]) as y`, where the same label can be captured by multiple accumulators.

Formally, an automaton A is a tuple $\langle Q, Q^{\mathbf{init}}, Q^{\mathbf{fin}}, T\rangle$ where

- Q is a finite set of *states*,
- $Q^{\mathbf{init}} \subseteq Q$ is a set of *initial states*,
- $Q^{\mathbf{fin}} \subseteq Q$ is a set of *final states*, and
- T is a set of *transition rules* of the form $q \xrightarrow{l} q$ where l is either $X : L[q]$ or ϵ.[1]

We also write these components by $\mathbf{st}(A)$, $\mathbf{init}(A)$, $\mathbf{fin}(A)$, and $\mathbf{tr}(A)$, respectively. Note that we allow ϵ-transitions. They are crucial in ambiguity test since elimination of ϵ-transitions does not preserve ambiguity (Section 6). We say an automaton to be ϵ-free if it has no ϵ-transition. We omit "X :" from a transition rule in the context where X does not matter.

[1] Note that each transition of the form $q_1 \xrightarrow{X:L[q_2]} q_3$ goes from one state q_1 to two states q_2 and q_3. Therefore our automata can be seen as a variation of binary (nondeterministic) top-down tree automata.

We define linearity for automata. For an automaton A, let $\mathbf{bv}_A(q)$ be the union of all the variable sets associated to the transitions reachable from q. Then, we require the automaton to be linear, that is, for each transition $q_1 \xrightarrow{X:L[q_3]} q_2$, it has to be that $(X \cup \mathbf{bv}_A(q_2)) \cap \mathbf{bv}_A(q_3) = \emptyset$. Any linear pattern will be translated to a linear automaton. Note, however, that some non-linear patterns are translated to linear automata, e.g.,

```
(a[] as x), (b[] as x).
```

In other words, we can safely relax our definition of linearity for patterns so that variables can capture non-consecutive sequences. This approach has been taken by CDuce [BCF02].

We below present two semantics, one used for type inference for variables and the other for ambiguity check.

In the first semantics, we formalize the behavior of tree automata with sequence accumulators informally described above. However, rather than directly dealing with sequence accumulation, we actually take two steps for the ease of reasoning. First, an automaton annotates each node of the input value with a set of variables while matching the value. Then, we extract a value for each variable by erasing the nodes whose annotated sets do not contain the variable. Annotated values ρ are values where each label is annotated with a set of variables and defined by the following syntax.

$$\begin{array}{rcll} \rho & ::= & a[\rho]^X & \text{annotated label} \\ & & \rho, \rho & \text{concatenation} \\ & & \epsilon & \text{empty sequence} \end{array}$$

Again, we omit "X" from an annotated label in the context where it does not matter. Now, the semantics is described by the matching relation $A \vdash v \in q \Rightarrow \rho$, read "in automaton A, value v is accepted by state q and yields annotated value ρ," by the following set of rules.

$$\frac{q_1 \in \mathbf{fin}(A)}{A \vdash \epsilon \in q_1 \Rightarrow \epsilon} \quad \text{(T-Fin)}$$

$$\frac{q_1 \xrightarrow{X:L[q_3]} q_2 \in \mathbf{tr}(A) \qquad a \in L \qquad A \vdash v_1 \in q_3 \Rightarrow \rho_1 \qquad A \vdash v_2 \in q_2 \Rightarrow \rho_2}{A \vdash a[v_1], v_2 \in q_1 \Rightarrow a[\rho_1]^X, \rho_2} \quad \text{(T-Lab)}$$

$$\frac{q_1 \xrightarrow{\epsilon} q_2 \in \mathbf{tr}(A) \qquad A \vdash v_1 \in q_2 \Rightarrow \rho_1}{A \vdash v_1 \in q_1 \Rightarrow \rho_1} \quad \text{(T-Eps)}$$

We write $A \vdash v \Rightarrow \rho$ when $A \vdash v \in q_0 \Rightarrow \rho$ with $q_0 \in \mathbf{init}(A)$, and write $A \vdash v$ when $A \vdash v \Rightarrow \rho$ for some ρ. We obtain a binding of each variable resulted from

the matching by using the function **env**, defined as follow.

$$\begin{array}{lcll}
\mathbf{env}(\epsilon)(x) & = & \epsilon & \\
\mathbf{env}(a[\rho_1]^X, \rho_2)(x) & = & a[\mathbf{erase}(\rho_1)], \rho_2(x) & \text{if } x \in X \\
\mathbf{env}(a[\rho_1]^X, \rho_2)(x) & = & \mathbf{env}(\rho_1)(x) & \text{if } x \notin X,\ \mathbf{env}(\rho_1)(x) \neq \epsilon \\
\mathbf{env}(a[\rho_1]^X, \rho_2)(x) & = & \mathbf{env}(\rho_2)(x) & \text{if } x \notin X,\ \mathbf{env}(\rho_1)(x) = \epsilon
\end{array}$$

where **erase** removes all the variable-set annotations from a given annotated value:

$$\begin{array}{lcl}
\mathbf{erase}(a[\rho]^X) & = & a[\mathbf{erase}(\rho)] \\
\mathbf{erase}(\rho_1, \rho_2) & = & \mathbf{erase}(\rho_1), \mathbf{erase}(\rho_2) \\
\mathbf{erase}(\epsilon) & = & \epsilon
\end{array}$$

To understand the definition of **env**, first recall that linearity ensures that, in any annotated value resulted from a matching, the same variables occur only in the same sequence. For example, we may have an annotated value

$$b[a[]^{\{\mathtt{x}\}}, a[]^{\{\mathtt{x}\}}]^{\emptyset}$$

but never

$$b[a[]^{\{\mathtt{x}\}}]^{\emptyset}, a[]^{\{\mathtt{x}\}}.$$

Thus, when the **env** function visits each node of the given annotated value, there are only three cases that the function has to do. When the node's variable set contains x, we retain this node (by erasing all the annotations in its content ρ_1) and proceed to the remaining sequence ρ_2. Otherwise, we skip the node and then traverse ρ_1. If the result of this traversal is not an empty sequence, then it means that x occurs in ρ_1 and therefore does not in ρ_2; so we ignore ρ_2. On the other hand, if the result *is* an empty sequence, then it means that x does not occur in ρ_1 and therefore it may in ρ_2; so we proceed to ρ_2.

The second semantics is the same as the first one except that it now yields a "path" (instead of an annotated value), which keeps track of all the states that the automaton followed during the matching. We will later define that an automaton is ambiguous if it can yield two different paths. Formally, *paths*, ranged over by π, ϕ, and ψ, are defined by the following syntax.

$$\begin{array}{lcl}
\pi, \phi, \psi & ::= & q[\pi], \pi \\
& & q, \pi \\
& & q
\end{array}$$

The "parsing" relation $A \vdash v \in q \Rightarrow \pi$ is defined by the following set of rules.

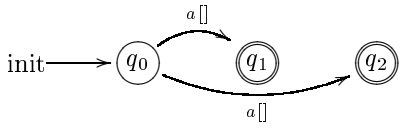
$$\frac{q_1 \in \mathbf{fin}(A)}{A \vdash \epsilon \in q_1 \Rightarrow q_1} \qquad \text{(TP-Fin)}$$

$$\frac{q_1 \xrightarrow{X:L[q_3]} q_2 \in \mathbf{tr}(A) \qquad a \in L \qquad A \vdash v_1 \in q_3 \Rightarrow \pi_1 \qquad A \vdash v_2 \in q_2 \Rightarrow \pi_2}{A \vdash a[v_1], v_2 \in q_1 \Rightarrow q_1[\pi_1], \pi_2} \qquad \text{(TP-Lab)}$$

$$\frac{q_1 \xrightarrow{\epsilon} q_2 \in \mathbf{tr}(A) \qquad A \vdash v \in q_2 \Rightarrow \pi_2}{A \vdash v \in q_1 \Rightarrow q_1, \pi_2} \qquad \text{(TP-EPS)}$$

We write $A \vdash v \Rightarrow \pi$ when $A \vdash v \in q \Rightarrow \pi$ with $q \in \mathbf{init}(A)$. We say that A is *ambiguous* for v if $A \vdash v \Rightarrow \pi$ and $A \vdash v \Rightarrow \pi'$ with $\pi \neq \pi'$, and that A is ambiguous with respect to another automaton B if A is ambiguous for some v such that $B \vdash v$.

We can construct an algorithm that translates a pattern to an automaton with variables that yield the same bindings as the pattern with the same ambiguity. The translation is obtained by modifying Thompson's algorithm for automata construction from string regular expressions [HU79] so as to treat label classes, contents of labels, and variables. Since the algorithm is quite standard, we give it in Appendix A. However, it is important to note that not all automata construction algorithms preserve ambiguity. For example, from the trivially unambiguous pattern `a[]`, a construction algorithm may produce the following automaton



which is ambiguous.

In the rest of this section, we present two standard operations on tree automata (in a slight variation), elimination of useless states and elimination of epsilon transitions. These will be used in the algorithms for ambiguity check and type inference.

4.2 Useless States Elimination

This operation removes, from a given automaton, all the states that are not reachable from the initial states and those that are not matched by any value. We define $\mathbf{reach}_A(q)$ as the set of states reachable from q and $B = \emptyset\text{-}\mathbf{elim}(A)$ as follows.

$$\begin{array}{lcl}
\mathbf{st}(B) & = & \{q \mid q \in \mathbf{reach}_A(\mathbf{init}(A)),\ \exists v.\ A \vdash v \in q\} \\
\mathbf{init}(B) & = & \mathbf{init}(A) \cap \mathbf{st}(B) \\
\mathbf{fin}(B) & = & \mathbf{fin}(A) \cap \mathbf{st}(B) \\
\mathbf{tr}(B) & = & \{q_1 \xrightarrow{X:L[q_2]} q_3 \in \mathbf{tr}(A) \mid q_1, q_2, q_3 \in \mathbf{st}(B), L \neq \emptyset\}
\end{array}$$

There is a well-known linear-time algorithm for useless states elimination [CDG+99]. We can easily prove the following expected properties.

4.1 Lemma: Let $B = \emptyset\text{-}\mathbf{elim}(A)$. Then, $A \vdash v \Rightarrow \rho$ if and only $B \vdash v \Rightarrow \rho$. Moreover, $A \vdash v \Rightarrow \pi$ if and only $B \vdash v \Rightarrow \pi$.

Proof: The result follows from a stronger statement: $A \vdash v \in q \Rightarrow \rho$ with $q \in \mathbf{reach}_A(\mathbf{init}(A))$ if and only if $B \vdash v \in q \Rightarrow \rho$. The proof can be done by straightforward induction on the derivation. The second statement of the lemma can also be proved in a similar way. ∎

4.3 Epsilon Elimination

This operation is exactly the same as the usual epsilon elimination. That is, we first define the function ϵ-**clos** to compute the set of states reachable by ϵ-transitions from a given state:

$$\epsilon\text{-}\mathbf{clos}_A(q_1) = \{q_2 \mid q_1 \xrightarrow{\epsilon}{}^{*} q_2 \in \mathbf{tr}(A)\}$$

($q_1 \xrightarrow{\epsilon}{}^{*} q_n \in \mathbf{tr}(A)$ means $q_i \xrightarrow{\epsilon} q_{i+1} \in \mathbf{tr}(A)$ for all $i = 1, \ldots, n-1$ with $n \geq 1$.) Then, we define $B = \epsilon\text{-}\mathbf{elim}(A)$ as follows.

$$\begin{array}{lcl}
\mathbf{st}(B) & = & \mathbf{st}(A) \\
\mathbf{init}(B) & = & \mathbf{init}(A) \\
\mathbf{fin}(B) & = & \{q_1 \in \mathbf{st}(A) \mid \epsilon\text{-}\mathbf{clos}_A(q_1) \cap \mathbf{fin}(A) \neq \emptyset\} \\
\mathbf{tr}(B) & = & \{q_1 \xrightarrow{X:L[q_3]} q_2 \mid q_4 \in \epsilon\text{-}\mathbf{clos}_A(q_1),\ q_4 \xrightarrow{X:L[q_3]} q_2 \in \mathbf{tr}(A)\}
\end{array}$$

4.2 Lemma: Let $B = \epsilon\text{-}\mathbf{elim}(A)$. Then, $A \vdash v \Rightarrow \rho$ if and only $B \vdash v \Rightarrow \rho$.

Proof: We show the lemma by a stronger statement: $q \xrightarrow{\epsilon}{}^{*} q' \in \mathbf{tr}(A)$ and $A \vdash v \in q' \Rightarrow \rho$ for some q' if and only if $B \vdash v \in q \Rightarrow \rho$. Both directions can be proved by straightforward induction on the derivation. ∎

5 Type Inference Algorithm

This section presents our algorithm for inferring types for pattern variables. Given an input type A and a pattern B (both represented by tree automata defined in Section 4), the inference algorithm works in two steps. First, we specialize B with respect to A such that the resulting automaton C behaves exactly the same as B except that it accepts only values from A. For this, we use product construction where we preserve the variable binding behavior in B. Second, we obtain, for each variable x, an automaton D_x such that D_x accepts a value v if and only if C accepts some value from A and yields a binding of x to v. We compute this automaton from C by retaining all transitions with the annotated set containing x and eliminating all the other.

Formally, given ϵ-free automata A and B, we first take the product of them and then eliminate useless states. (Note that ϵ-elimination preserves the binding behavior.)

$$C = \emptyset\text{-}\mathbf{elim}(A \times B)$$

Here, $A \times B$ is defined as follows.

$$
\begin{array}{rcl}
\mathbf{st}(A \times B) & = & \mathbf{st}(A) \times \mathbf{st}(B) \\
\mathbf{init}(A \times B) & = & \mathbf{init}(A) \times \mathbf{init}(B) \\
\mathbf{fin}(A \times B) & = & \mathbf{fin}(A) \times \mathbf{fin}(B) \\
\mathbf{tr}(A \times B) & = & \left\{ \langle p_1, q_1 \rangle \xrightarrow{X:(K \cap L)[\langle p_3, q_3 \rangle]} \langle p_2, q_2 \rangle \;\middle|\; \begin{array}{l} K \cap L \neq \emptyset \\ p_1 \xrightarrow{K[p_3]} p_2 \in \mathbf{tr}(A), \\ q_1 \xrightarrow{X:L[q_3]} q_2 \in \mathbf{tr}(B) \end{array} \right\}
\end{array}
$$

Note that, unlike usual product construction, for each pair of transitions (one from A and the other from B), we take the intersection of the label classes of the two transitions, and that we retain the set of variables in the transition from B.

We then obtain the following automaton D_x for each variable x.

$$
\begin{array}{rcl}
\mathbf{st}(D_x) & = & \{r, \hat{r} \mid r \in \mathbf{st}(C)\} \\
\mathbf{init}(D_x) & = & \{\hat{r} \mid r \in \mathbf{init}(C)\} \\
\mathbf{fin}(D_x) & = & \{r, \hat{r} \mid r \in \mathbf{fin}(C)\} \\
\mathbf{tr}(D_x) & = & \left\{ \hat{r_1} \xrightarrow{L[r_3]} \hat{r_2} \;\middle|\; r_1 \xrightarrow{X:L[r_3]} r_2 \in \mathbf{tr}(C),\ x \in X \right\} \\
& \cup & \left\{ \hat{r_1} \xrightarrow{\epsilon} \hat{r_2} \;\middle|\; r_1 \xrightarrow{X:L[r_3]} r_2 \in \mathbf{tr}(C),\ x \notin X,\ x \in \mathbf{bv}(r_2) \right\} \\
& \cup & \left\{ \hat{r_1} \xrightarrow{\epsilon} \hat{r_3} \;\middle|\; r_1 \xrightarrow{X:L[r_3]} r_2 \in \mathbf{tr}(C),\ x \notin X,\ x \notin \mathbf{bv}(r_2) \right\} \\
& \cup & \left\{ r_1 \xrightarrow{L[r_3]} r_2 \;\middle|\; r_1 \xrightarrow{X:L[r_3]} r_2 \in \mathbf{tr}(C) \right\}
\end{array}
$$

The automaton D_x has exactly twice as many states as the automaton C, i.e., it has states r_i and $\hat{r_i}$ for each state r_i in C. Intuitively, r_i in D_x accepts exactly the same set of values as r_i in C, whereas $\hat{r_i}$ in D_x accepts the set of values to which r_i in C binds x. Therefore, in the construction of D_x, we copy all the transitions from C to D_x relevant to the first kind of states. For the second kind, we create a transition in D_x from each transition $r_1 \xrightarrow{X:L[r_3]} r_2$ in C as follows. First, if X contains x, then we create a transition $\hat{r_1} \xrightarrow{L[r_3]} \hat{r_2}$ in D_x, where, for the remainder sequence, the transition goes to $\hat{r_2}$ since there may be variables there, whereas, for the content, it goes to r_3 since we want to simply capture the whole value matched by r_3. Second, if X does not contain x, then x may occur in r_2, r_3, or neither; recall that linearity ensures x to never occur in both. Therefore, we create an ϵ-transition from $\hat{r_1}$ to $\hat{r_2}$ if x occurs in r_2, and one from $\hat{r_1}$ to $\hat{r_3}$ otherwise.

We can prove that the algorithm is sound and complete. We first show the expected properties of product construction and a technical lemma to characterize the first kind of states. The main theorems follows after these.

5.1 Lemma: $A \vdash v \in p_1$ and $B \vdash v \in q_1 \Rightarrow \rho$ if and only if $A \times B \vdash v \in \langle p_1, q_1 \rangle \Rightarrow \rho$.

Proof: Both directions can be proved by straightforward induction on the derivation(s). ∎

5.2 Lemma: For each x, if $C \vdash v \in r_1$, then $D_x \vdash v \in r_1$, and vice versa.

Proof: Both directions can be proved by a straightforward induction on the derivation. ∎

5.3 Theorem [Soundness]: If $D_x \vdash w$, then $C \vdash v \Rightarrow \rho$ and $w = \mathbf{env}(\rho)(x)$ for some v, ρ.

Proof: We prove a stronger statement

> if $D_x \vdash w \in \hat{r_1}$, then $C \vdash v \in r_1 \Rightarrow \rho$ and $w = \mathbf{env}(\rho)(x)$ for some v, ρ.

by induction on the derivation of $D_x \vdash w \in \hat{r_1}$.

Case T-FIN: $\quad \hat{r_1} \in \mathbf{fin}(D_x) \qquad w = \epsilon$

By the definition of Γ, we have $r_1 \in \mathbf{fin}(C)$. The result follows from T-FIN.

Case T-LAB: $\quad w = a[w_1], w_2 \qquad \hat{r_1} \xrightarrow{L[r_4]} \hat{r_3} \in \mathbf{tr}(D_x)$
$\qquad\qquad\quad D_x \vdash v_1 \in r_4 \qquad D_x \vdash v_2 \in \hat{r_3}$

By the the induction hypothesis, $C \vdash v_2 \in r_3 \Rightarrow \rho_2$ with $w_2 = \mathbf{env}(\rho_2)(x)$. Also, $\hat{r_1} \xrightarrow{L[r_4]} \hat{r_3} \in \mathbf{tr}(D_x)$ implies that $r_1 \xrightarrow{X:L[r_4]} r_3 \in \mathbf{tr}(C)$ and $x \in X$. By Lemma 5.2, $C \vdash w_1 \in r_4 \Rightarrow \rho_1$ for some ρ_1. From these and T-LAB, we obtain $C \vdash a[w_1], v_2 \in r_1 \Rightarrow a[\rho_1]^X, \rho_2$. Further, $\mathbf{env}(a[\rho_1]^X, \rho_2) = a[\mathbf{erase}(\rho_1)], \mathbf{env}(\rho_2)(x) = a[w_1], w_2 = w$. The result follows.

Case T-EPS: $\quad \hat{r_1} \xrightarrow{\epsilon} \hat{r_3} \in \mathbf{tr}(D_x) \qquad D_x \vdash w \in \hat{r_3}$

By the the induction hypothesis, $C \vdash v_2 \in r_3 \Rightarrow \rho_2$ with $w = \mathbf{env}(\rho_2)(x)$ for some v_2, ρ_2. We have two subcases.

- $r_1 \xrightarrow{X:L[r_4]} r_3 \in \mathbf{tr}(C)$ and $x \notin X$ with $x \notin \mathbf{bv}(r_4)$. From the definition of C, there are v_1, ρ_1 such that $C \vdash v_1 \in r_4 \Rightarrow \rho_1$. In addition, take some a from L. Then, by T-LAB, we obtain $C \vdash a[v_1], v_2 \in r_1 \Rightarrow a[\rho_1]^X, \rho_2$. Further, from $x \notin \mathbf{bv}(r_4)$, we have $x \notin \mathbf{bv}(\rho_1)$ and therefore $\mathbf{env}(a[\rho_1]^X, \rho_2)(x) = \mathbf{env}(\rho_2)(x) = w$.

- $r_1 \xrightarrow{X:L[r_3]} r_4 \in \mathbf{tr}(C)$ and $x \notin X$ with $x \in \mathbf{bv}(r_3)$. From the definition of C, there are v_1, ρ_1 such that $C \vdash v_1 \in r_4 \Rightarrow \rho_1$. In addition, take some a from L. Then, by T-LAB, we obtain $C \vdash a[v_2], v_1 \in r_1 \Rightarrow a[\rho_2]^X, \rho_1$. We have further two subcases.

  - When $x \in \mathbf{bv}(\rho_2)$, we have $\mathbf{env}(a[\rho_2]^X, \rho_1)(x) = \mathbf{env}(\rho_2)(x) = w$. The result follows.
  - When $x \notin \mathbf{bv}(\rho_2)$, we have $\mathbf{env}(a[\rho_2]^X, \rho_1)(x) = \mathbf{env}(\rho_1)(x)$. Further, since $x \in \mathbf{bv}(r_3)$ implies $x \notin \mathbf{bv}(r_4)$ by the variable restriction, we have $x \notin \mathbf{bv}(\rho_1)$ and therefore $\mathbf{env}(\rho_1)(x) = \epsilon$. On the other hand, $x \notin \mathbf{bv}(\rho_2)$ implies $w = \mathbf{env}(\rho_2)(x) = \epsilon$. The result follows. ∎

5.4 Theorem [Completeness]: If $C \vdash v \Rightarrow \rho$ and $w = \mathbf{env}(\rho)(x)$ for some v, ρ, then $D_x \vdash w$.

Proof: We prove a stronger statement

> If $C \vdash v \in r_1 \Rightarrow \rho$ and $w = \mathbf{env}(\rho)(x)$ for some v, ρ, then $D_x \vdash w \in \hat{r_1}$.

by induction on the derivation of $C \vdash v \in r_1 \Rightarrow \rho$.

Case T-FIN: $r_1 \in \mathbf{fin}(C) \qquad v = \rho = \epsilon$

The result immediately holds since $\hat{r_2} \in \mathbf{fin}(D_x)$.

Case T-LAB: $r_1 \xrightarrow{X:L[r_4]} r_3 \in \mathbf{tr}(D_x) \qquad v = a[v_1], v_2 \qquad \rho = a[\rho_1]^X, \rho_2$
$a \in L \qquad C \vdash v_1 \in r_4 \Rightarrow \rho_1 \qquad C \vdash v_2 \in r_3 \Rightarrow \rho_2$

By the the induction hypothesis, we obtain the following.

$$D_x \vdash \mathbf{env}(\rho_2)(x) \in \hat{r_3} \tag{1}$$
$$D_x \vdash \mathbf{env}(\rho_1)(x) \in \hat{r_4} \tag{2}$$

We have three cases.

- $x \in X$. This implies $w = a[\mathbf{erase}(\rho_1)], \mathbf{env}(\rho_2)(x)$. Also, $\hat{r_1} \xrightarrow{L[r_4]} \hat{r_3} \in \mathbf{tr}(D_x)$. By Lemma 5.2, $D_x \vdash \mathbf{erase}(\rho_1) \in r_4$. With (1) and T-LAB, the result follows.
- $x \notin X$ and $x \in \mathbf{bv}(\rho_1)$. This implies $w = \mathbf{env}(\rho_1)(x)$. Also, $x \in \mathbf{bv}(r_4)$ and therefore $\hat{r_1} \xrightarrow{\epsilon} \hat{r_4} \in \mathbf{tr}(D_x)$. With (2) and T-EPS, the result follows.
- $x \notin X$ and $x \notin \mathbf{bv}(\rho_1)$. This implies $w = \mathbf{env}(\rho_2)(x)$. We have further two subcases.
  - $x \notin \mathbf{bv}(r_4)$. Then, $\hat{r_1} \xrightarrow{\epsilon} \hat{r_3} \in \mathbf{tr}(D_x)$ and therefore the result follows from (1) and T-EPS.
  - $x \in \mathbf{bv}(r_4)$. Then, $x \notin \mathbf{bv}(r_3)$ and therefore $x \notin \mathbf{bv}(\rho_2)$, implying $w = \mathbf{env}(\rho_2)(x) = \epsilon$. Also, $\hat{r_1} \xrightarrow{\epsilon} \hat{r_4} \in \mathbf{tr}(D_x)$. On the other hand, from $x \notin \mathbf{bv}(\rho_1)$, we have $\mathbf{env}(\rho_1)(x) = \epsilon$. The result follows from (2) and T-EPS. ∎

6 Ambiguity check algorithm

This section presents our algorithm for checking ambiguity for tree automata defined in Section 4.

Although our goal is ambiguity for tree automaton with ϵ-transitions with respect to a given input type (represented by another tree automaton), let us first consider a much more simplified setting: ambiguity for string automata with no ϵ-transition and no input type. Concretely, given a string automaton

B, check whether there are two different accepting paths (i.e., a path from an initial state to a final state) that spell out the same string.

The simplest algorithm for it would be the following.[2]

1. Take the product $C = B \times B$;
2. Eliminate all the useless states from C;
3. Answer "ambiguous" if and only if there is a state $\langle q, q' \rangle$ with $q \neq q'$.

We can easily see why it is correct by observing that if there are two different accepting paths in B

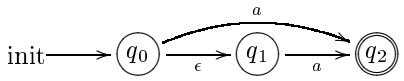
$$q_1 \dots q_n \quad q'_1 \dots q'_n$$

both spelling out a string s, then there is an accepting path

$$\langle q_1, q'_1 \rangle \dots \langle q_n, q'_n \rangle$$

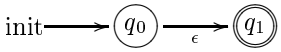
spelling out s where $q_i \neq q'_i$ for some i, and vice versa. (Note that $\langle q_i, q'_i \rangle$ is not useless in C since it is in the accepting path.)

The algorithm becomes more complicated if the given automaton may contain ϵ-transitions. First, we cannot just eliminate those transitions since this does not necessarily preserve ambiguity. For example, the following automaton is ambiguous, but ϵ-elimination makes it unambiguous.

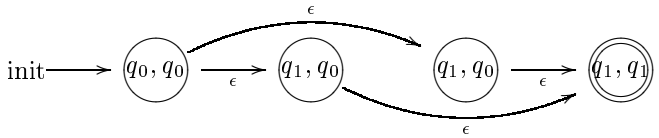


(This automaton is never produced by the translation in Appendix A, but we can easily find an ambiguous automaton resulted from the translation whose ϵ-elimination is unambiguous, e.g., `a**`.)

Therefore we directly treat ϵ-transitions in our ambiguity checking algorithm. We would like to obtain the algorithm based on the one above, but then what is product construction with ϵ-transitions? It is wrong to just create a transition $\langle q_1, q_2 \rangle \xrightarrow{\epsilon} \langle q_3, q_4 \rangle$ whenever there are transitions $q_1 \xrightarrow{\epsilon} q_2$ and $q_3 \xrightarrow{\epsilon} q_4$. As a counter example, if we take the product of the above automaton and itself, then we obtain exactly the same as the original one except the states each become $\langle q_0, q_0 \rangle$, $\langle q_1, q_1 \rangle$, and $\langle q_2, q_2 \rangle$, which does not respect the fact that the two transitions with a are "racing." The right way of product construction is therefore to create an ϵ-transition whenever there is an ϵ-transition in either of the original automata. For example, the product of the following automaton and itself



is the following.



[2] This algorithm was suggested by Sanjeev Kanna.

The subtlety is that we cannot judge unambiguity by whether each resulting state is a pair of the same state in the original automaton. For example, the input automaton above is unambiguous, but the resulting automaton contains the states $\langle q_1, q_0 \rangle$ and $\langle q_0, q_1 \rangle$, which are pairs of different states in B. Note, however, that these states arise from the fact that the ϵ-transition from $\langle q_0, q_0 \rangle$ to $\langle q_1, q_0 \rangle$ occurs by moving from q_0 to q_1 in the first component of the product and staying in q_0 in the second component; similarly for the other transitions. From this observation, we can know that there is actually only one path from q_0 to q_1: check that, for each path from $\langle q_0, q_0 \rangle$ to $\langle q_1, q_1 \rangle$ in C, the sequence of the states in the first component is the same as that of the states in the second component *modulo* consecutive duplicates. One small care is needed: consecutive duplicates may arise from a cyclic ϵ-transition from one state to the same. However, since this immediately means ambiguity, we can eliminate this possibility in the first place.

The remaining is to extend the algorithm for tree automata and input types. The first is straightforward: we can simply replace the product construction and useless states elimination for string automata by the corresponding ones for tree automata. (Useless states elimination is already presented in Section 4.2. For product construction, we need a slight modification of the one in Section 5 to treat ϵ-transitions.) For the second, we use the same technique as used in Section 5: take another product construction of the input type and the above self-product.

Formally, let us define the set of *pivot states* of an automaton B

$$\mathbf{pivot}(B) = \{q_1, q_2, q_3 \mid q_1 \xrightarrow{L[q_2]} q_3 \in \mathbf{tr}(B)\} \cup \mathbf{init}(B) \cup \mathbf{fin}(B).$$

Also, define

$$\begin{array}{lcl} \mathbf{dupelim}(\epsilon) & = & \epsilon \\ \mathbf{dupelim}(q_1 q_2 \ldots q_n) & = & \left\{ \begin{array}{ll} \mathbf{dupelim}(q_2 \ldots q_n) & (q_1 = q_2) \\ q_1 \mathbf{dupelim}(q_2 \ldots q_n) & (q_1 \neq q_2). \end{array} \right. \end{array}$$

Given automata A and B where A is ϵ-free, we first compute

$$D = \emptyset\text{-}\mathbf{elim}(A \times B \times B).$$

Here, $A \times B$ is defined by:

$$\begin{array}{lcl} \mathbf{st}(A \times B) & = & \mathbf{st}(A) \times \mathbf{st}(B) \\ \mathbf{init}(A \times B) & = & \mathbf{init}(A) \times \mathbf{init}(B) \\ \mathbf{fin}(A \times B) & = & \mathbf{fin}(A) \times \mathbf{fin}(B) \\ \mathbf{tr}(A \times B) & = & \left\{ \langle p_1, q_1 \rangle \xrightarrow{X:(K \cap L)[\langle p_3, q_3 \rangle]} \langle p_2, q_2 \rangle \;\middle|\; \begin{array}{l} K \cap L \neq \emptyset \\ p_1 \xrightarrow{K[p_3]} p_2 \in \mathbf{tr}(A), \\ q_1 \xrightarrow{X:L[q_3]} q_2 \in \mathbf{tr}(B) \end{array} \right\} \\ & & \cup \left\{ \langle p, q_1 \rangle \xrightarrow{\epsilon} \langle p, q_2 \rangle \mid p \in \mathbf{st}(A),\ q_1 \xrightarrow{\epsilon} q_2 \in \mathbf{tr}(B) \right\} \end{array}$$

For $\langle p, q, r \rangle \in \mathbf{pivot}(D)$, we define the predicate $\mathbf{ambchk}(\langle p, q, r \rangle)$ as either of the following holds.

1. $q \neq r$
2. For some $\langle p_i, q_i, r_i\rangle \xrightarrow{\epsilon} \langle p_{i+1}, q_{i+1}, r_{i+1}\rangle \in \mathbf{tr}(D)$ $(i = 1, \ldots, n-1)$ where $\langle p_1, q_1, r_1\rangle = \langle p, q, r\rangle$ and $\langle p_n, q_n, r_n\rangle \in \mathbf{pivot}(D)$, either
   (a) $\langle p_i, q_i, r_i\rangle = \langle p_{i+1}, q_{i+1}, r_{i+1}\rangle$ for some i, or
   (b) $\mathbf{dupelim}(q_1 \ldots q_n) \neq \mathbf{dupelim}(r_1 \ldots r_n)$.

We then answer "ambiguous" if and only if $\mathbf{ambchk}(\langle p, q, r\rangle)$ holds for all $\langle p, q, r\rangle \in \mathbf{pivot}(D)$.

We can prove that the algorithm is both sound and complete. The main theorems follow after a technical lemma on product construction.

6.1 Lemma: $A \vdash v \in p_1$ and $B \vdash v \in q_1 \Rightarrow \pi$ for some π if and only if $A \times B \vdash v \in \langle p_1, q_1\rangle \Rightarrow \psi$ for some ψ.

Proof: Both directions can be proved by straightforward induction on the derivation(s). ∎

6.2 Theorem [Soundness]: If the above algorithm answers "ambiguous," then B is ambiguous w.r.t. A.

Proof: We prove a stronger statement: for all $\langle p, q, r\rangle \in \mathbf{pivot}(D)$, if there is $\langle p', q', r'\rangle \in \mathbf{reach}_D(\langle p, q, r\rangle) \cap \mathbf{pivot}(D)$ that satisfies $\mathbf{ambchk}(\langle p, q, r\rangle)$, then $A \vdash v \in p$ with $B \vdash v \in q \Rightarrow \phi$ and $B \vdash v \in r \Rightarrow \psi$ where $\phi \neq \psi$. Since $\langle p', q', r'\rangle$ is reachable from $\langle p, q, r\rangle$, there are

$$(\langle p, q, r\rangle =)\ \langle p_1, q_1, r_1\rangle, \ldots, \langle p_n, q_n, r_n\rangle\ (= \langle p', q', r'\rangle) \in \mathbf{pivot}(D)$$

such that, for each $i = 1, \ldots, n-1$, either

A $\langle p_i, q_i, r_i\rangle \xrightarrow{\epsilon}{}^{+} \langle p_{i+1}, q_{i+1}, r_{i+1}\rangle \in \mathbf{tr}(D)$,

B $\langle p_i, q_i, r_i\rangle \xrightarrow{L[\langle p_{i+1}, q_{i+1}, r_{i+1}\rangle]} \langle p'', q'', r''\rangle \in \mathbf{tr}(D)$ for some $\langle p'', q'', r''\rangle$, or

C $\langle p_i, q_i, r_i\rangle \xrightarrow{L[\langle p'', q'', r''\rangle]} \langle p_{i+1}, q_{i+1}, r_{i+1}\rangle \in \mathbf{tr}(D)$ for some $\langle p'', q'', r''\rangle$.

The proof proceeds by induction on n.

In the case $n = 1$, we have two cases.

- (1) holds. Since D is ϵ-free, $\langle p, q, r\rangle$ accepts some value v, i.e., $D \vdash v \in \langle p, q, r\rangle$. Therefore we have, by Lemma 6.1, $A \vdash v \in p$ with $B \vdash v \in q \Rightarrow \phi$ and $B \vdash v \in r \Rightarrow \psi$. Since $q \neq r$, we obtain $\phi \neq \psi$.
- (2) holds. First, there are

  $$\langle p, q'_j, r'_j\rangle \xrightarrow{\epsilon} \langle p, q'_{j+1}, r'_{j+1}\rangle \in \mathbf{tr}(D)\ \ (j = 1, \ldots, k-1)$$

  where $\langle p, q'_1, r'_1\rangle = \langle p, q, r\rangle$ and $\langle p, q'_k, r'_k\rangle \in \mathbf{pivot}(D)$. Since $\langle p, q'_k, r'_k\rangle$ accepts some value v, we know, by Lemma 6.1, that $A \vdash v \in p$ with

$B \vdash v \in q'_k \Rightarrow \phi$ and $B \vdash v \in r'_k \Rightarrow \psi$. Therefore $B \vdash v \in q'_k \Rightarrow \phi'\phi$ and $B \vdash v \in r'_k \Rightarrow \psi'\psi$ with $\mathbf{dupelim}(\phi') = \mathbf{dupelim}(q'_1 \ldots q'_k)$ and $\mathbf{dupelim}(\psi') = \mathbf{dupelim}(r'_1 \ldots r'_k)$. If (2a) holds, i.e., $\langle p, q'_j, r'_j \rangle = \langle p, q'_{j+1}, r'_{j+1} \rangle$ for some j, then either $q'_j \xrightarrow{\epsilon} q'_j \in \mathbf{tr}(B)$ or $r'_j \xrightarrow{\epsilon} r'_j \in \mathbf{tr}(B)$. In the first case, we can clearly find ϕ'' such that $B \vdash v \in q'_1 \Rightarrow \phi''\phi$ where $\phi' \neq \phi''$ and $\mathbf{dupelim}(\phi') = \mathbf{dupelim}(\phi'')$; similarly for the second case. If (2a) does not hold, i.e., $\langle p, q'_j, r'_j \rangle \neq \langle p, q'_{j+1}, r'_{j+1} \rangle$ for any j, then $\mathbf{dupelim}(\psi') = \psi'$ and $\mathbf{dupelim}(\phi') = \phi'$. Moreover, since (2b) must hold, $\mathbf{dupelim}(q'_1 \ldots q'_k) \neq \mathbf{dupelim}(r'_1 \ldots r'_k)$; thereby the result follows.

In the case $n > 1$, there are three cases given above.

A By the induction hypothesis, $A \vdash v_2 \in p_2$ with $B \vdash v_2 \in q_2 \Rightarrow \psi_2$ and $B \vdash v_2 \in r_2 \Rightarrow \phi_2$ where $\psi_2 \neq \phi_2$, for some v_2, ψ_2, ϕ_2. From the definition of D, there are $p \xrightarrow{\epsilon}{}^{+} p_2 \in \mathbf{tr}(A)$ and $q \xrightarrow{\epsilon}{}^{+} q_2$, $r \xrightarrow{\epsilon}{}^{+} r_2 \in \mathbf{tr}(B)$. Therefore the result follows.

B By the induction hypothesis, $A \vdash v_2 \in p_2$ with $B \vdash v_2 \in q_2 \Rightarrow \psi_2$ and $B \vdash v_2 \in r_2 \Rightarrow \phi_2$ where $\psi_2 \neq \phi_2$, for some v_2, ψ_2, ϕ_2. In addition, since $\langle p'', q'', r'' \rangle$ accepts some value, we have $A \vdash v'' \in p''$ with $B \vdash v'' \in q'' \Rightarrow \psi''$ and $B \vdash v'' \in r'' \Rightarrow \phi''$, for some v'', ψ'', ϕ'' (where ψ'' and ϕ'' are possibly equal). Take some a from N and let $v = a[v_2], v''$ with $\phi = q[\phi_2], \phi''$ and $\psi = q[\psi_2], \psi''$. From the definition of D, there are $p \xrightarrow{K[p_2]} p'' \in \mathbf{tr}(A)$ and $q \xrightarrow{L[q_2]} q''$, $r \xrightarrow{M[r_2]} r'' \in \mathbf{tr}(B)$ where $N = K \cap L \cap M$. Therefore the result follows.

C Similar to B. ∎

6.3 Theorem [Completeness]: If B is ambiguous w.r.t. A, then the above algorithm answers "ambiguous."

Proof: We prove the result by showing a stronger statement: Let $A \vdash v \in p$ with $B \vdash v \in q \Rightarrow \phi$ and $B \vdash v \in r \Rightarrow \psi$ where $p \in \mathbf{reach}_A(\mathbf{init}(A))$ and $q = r \in \mathbf{pivot}(B) \cap \mathbf{reach}_B(\mathbf{init}(B))$. If $\phi \neq \psi$, then $\mathbf{ambchk}(\langle p, q, r \rangle)$. (The result follows from this since, when $q, r \in \mathbf{init}(B)$, either $q \neq r$ and (1) holds, or $q = r$ and the above statement implies the result.) The proof proceeds by simultaneous induction on the derivations of $A \vdash v \in p$, $B \vdash v \in q \Rightarrow \phi$, and $B \vdash v \in r \Rightarrow \psi$.

Case: $v = \epsilon$ $p \in \mathbf{fin}(A)$ $q, r \in \mathbf{fin}(B)$

This case never arises since $\phi = q = r = \psi$.

Case: $v = a[v''], v'$ $a \in K$ $a \in L$ $a \in M$
$p \xrightarrow{K[p'']} p' \in \mathbf{tr}(A)$ $q \xrightarrow{L[q'']} q'$, $r \xrightarrow{M[r'']} r' \in \mathbf{tr}(B)$
$A \vdash v' \in p'$ $A \vdash v'' \in p''$
$B \vdash v' \in q' \Rightarrow \phi'$ $B \vdash v'' \in q'' \Rightarrow \phi''$
$B \vdash v' \in r' \Rightarrow \psi'$ $B \vdash v'' \in r'' \Rightarrow \psi''$
$\phi = q[\phi''], \phi'$ $\psi = r[\psi''], \psi'$

From $\phi \neq \psi$, either $\phi'' \neq \psi''$ or $\phi' \neq \psi'$. Further, by the definition of D, there is

$$\langle p, q, r\rangle \xrightarrow{(K \cap L \cap M)[\langle p'', q'', r''\rangle]} \langle p', q', r'\rangle \in \mathbf{tr}(D).$$

If $q'' \neq r''$ or $q' \neq r'$, then (1) holds. Otherwise, the result follows by the induction hypothesis.

Case: $q_i \xrightarrow{\epsilon} q_{i+1} \in \mathbf{tr}(B)$ $(i = 1, \ldots, n-1)$
$r_i \xrightarrow{\epsilon} r_{i+1} \in \mathbf{tr}(B)$ $(i = 1, \ldots, m-1)$
$B \vdash v \in q_n \Rightarrow \phi'$ $\quad B \vdash v \in r_m \Rightarrow \psi'$
$q = q_1$ $\quad r = r_1$ $\quad q_n, r_m \in \mathbf{pivot}(B)$
$\phi = q_1 \ldots q_n \phi'$ $\quad \psi = r_1 \ldots r_m \psi'$

By the definition of D, we have $\langle p, q'_i, r'_i\rangle \xrightarrow{\epsilon} \langle p, q'_{i+1}, r'_{i+1}\rangle \in \mathbf{tr}(D)$ $(i = 1, \ldots, l-1)$ where $\mathbf{dupelim}(q_1 \ldots q_n) = \mathbf{dupelim}(q'_1 \ldots q'_l)$ and $\mathbf{dupelim}(r_1 \ldots r_m) = \mathbf{dupelim}(r'_1 \ldots r'_l)$. In addition, from $q_n, r_m \in \mathbf{pivot}(B)$, we have $\langle p, q_n, r_m\rangle \in \mathbf{pivot}(D)$.

Let us assume $q_n = r_m$ (otherwise (1) holds). From $\phi \neq \psi$, either $\phi' \neq \psi'$ or $q_1 \ldots q_n \neq r_1 \ldots r_m$. In the first case, the result follows by the induction hypothesis. In the second case, let assume $\langle p, q'_i, r'_i\rangle \neq \langle p, q'_{i+1}, r'_{i+1}\rangle$ for any i (otherwise (2a) holds). Then, we have $\mathbf{dupelim}(q_1 \ldots q_n) = q_1 \ldots q_n$ and $\mathbf{dupelim}(r_1 \ldots r_m) = r_1 \ldots r_m$ since otherwise, by the definition of D, we can find i such that $q'_i = q'_{i+1}$ and $r'_i = r'_{i+1}$. Therefore (2b) holds. ∎

7 Related work

Pattern matching is a popular idiom in designs of XML processing languages and can be found in a variety of styles. The design of pattern matching given in the present paper is derived from our previous proposal [HP02] by replacing the first-match semantics with the nondeterministic semantics plus the ambiguity check. The design based on the first-match has been further adopted and extended in the design of the CDuce language [BCF02] (in particular, they incorporate "shortest-match" pattern constructs). Since XDuce and CDuce are functional languages influenced by the ML family [MTH90, LVD+96, etc.], these use a single-match semantics of pattern matching, where a successful match yields just one binding. On the other hand, many query languages for XML use the *all-matches* style, which yields a *set* of bindings corresponding to all possible matches [DFF+98, AQM+97, CS98, CG00, NS00, Cla99, NS98, Mur97]. More comparisons of the expressiveness of various pattern matching designs can be found in [HP02].

Various forms of type inference problems have also been a hot topic in the area of static typing for XML. The inference algorithm presented in this paper can be seen as a variation of our previous algorithm for the first-match pattern matching (though the presentation is quite different). On one hand, the present algorithm is simpler in the sense that it does not need to treat "values already captured by the patterns of higher priority," which arises from the first-match semantics. On the other hand, the present algorithm improves the previous one

in the capability of the precise inference for pattern variables in non-tail positions. Other various type inference techniques can be found for query languages [MS99, PV00, MSV00, Toz01]. All of these techniques use product construction of automata like ours. However, their techniques do not contain our treatment of sequence-capturing pattern variables and the automata encoding using "sequence-accumulators." Finally, an inference algorithm similar to ours has independently been developed and presented by Frisch, Castagna, and Benzaken [FCB02] (their algorithm further treats more complex pattern constructs such as intersection patterns).

Ambiguity for string regular expressions and automata has been a classical question [BEGO71, BK93]. Using terminology in [BK93], our ambiguity can be categorized as *strong ambiguity*, which requires uniqueness of correspondence between subexpressions of the regular expression and subsequences of the given value (besides the fact that we treat trees rather than strings and takes input types into account). In the same paper, a checking procedure for strong ambiguity is given as a reduction to an ambiguity checking algorithm for LR(0) grammars. We are not aware of any presentation of any simple algorithm as ours directly operating on automata. Kawaguchi has developed an algorithm for detecting *weak ambiguity* for tree grammars (like us) [Kaw01], which requires uniqueness of correspondence between labels in the regular expression and labels in the value. Although his algorithm uses product construction for string automata many times and is quite complex, the essence seems to be similar to our algorithm (in fact, we believe that we can obtain an algorithm for checking weak ambiguity by a slight modification of our algorithm).

Vansummeren has recently reported yet another approach to regular expression pattern matching [Van03]. The paper raises an issue of a mismatch between a "normal" intuition behind the longest match and the simulation of the longest match by the first match and recursion. The proposed semantics deals with this by treating the first-match and the longest-match policies separately.

8 Conclusions

In this paper, we have presented a design of regular expression pattern matching. This design is an improvement of our previous proposal of pattern matching, where we drop the first-match priority rules from the semantics and instead perform an ambiguity check, thus making the behavior of pattern matching simpler and more intuitive. Furthermore, related to the pattern matching design, we have described two algorithms, type inference for pattern variables and ambiguity checking.

We have incorporated this design in the implementation of the XDuce system (available at `xduce.sourceforge.net`) [HP03]. One issue here is that the current language has a support for attribute types [HM02] in the style of RELAX NG schemas [CM01], which the algorithms in this paper do not deal with. We have already figured out how to extend the type inference for this feature and have implemented it. For the ambiguity checking, an extension is left for

future work.

APPENDIX

A Translation from patterns to tree automata with variables

In this section, we show an algorithm to translate a pattern to a tree automaton with variables. The translation not only produces an automaton yielding the same variable bindings as the original pattern, but also preserves its ambiguity.

Given a "starting" pattern $\mathtt{P}_0$ and a pattern definition E, we create an automaton $A^{\mathbf{all}}_{\mathtt{P}_0,\emptyset}$ where

$$A^{\mathbf{all}}_{\mathtt{P},X} = \left\langle \begin{array}{ll} Q_{E(\mathtt{P}),X} & \cup \bigcup_{\mathtt{Y}\in\mathbf{dom}(E)} (Q_{E(\mathtt{Y}),\emptyset} \cup \{q_{\mathtt{Y}}\}), \\ Q^{\mathbf{init}}_{\mathtt{P},X}, & \\ Q^{\mathbf{fin}}_{\mathtt{P},X} & \cup \bigcup_{\mathtt{Y}\in\mathbf{dom}(E)} Q^{\mathbf{fin}}_{E(\mathtt{Y}),\emptyset}, \\ T_{\mathtt{P},X} & \cup \bigcup_{\mathtt{Y}\in\mathbf{dom}(E)} (T_{E(\mathtt{Y}),\emptyset} \cup (\{q_{\mathtt{Y}}\} \xrightarrow{\epsilon} Q^{\mathbf{init}}_{E(\mathtt{Y}),\emptyset})) \end{array} \right\rangle$$

and $\langle Q_{\mathtt{P},X}, Q^{\mathbf{init}}_{\mathtt{P},X}, Q^{\mathbf{fin}}_{\mathtt{P},X}, T^{\mathbf{fin}}_{\mathtt{P},X}\rangle = A_{\mathtt{P},X}$ is inductively defined as follows.

$$\begin{array}{lcl} A_{\mathtt{()},X} & = & \langle \{q_1\},\ \{q_1\},\ \{q_1\},\ \emptyset\rangle \\ A_{\mathtt{L[Y]},X} & = & \langle \{q_1,q_2\},\ \{q_1\},\ \{q_2\},\ \{q_1 \xrightarrow{X:L[q_{\mathtt{Y}}]} q_2\}\rangle \\ A_{\mathtt{(P_1,P_2)},X} & = & \langle\ Q_{\mathtt{P}_1,X} \cup Q_{\mathtt{P}_2,X},\ Q^{\mathbf{init}}_{\mathtt{P}_1,X},\ Q^{\mathbf{fin}}_{\mathtt{P}_2,X}, \\ & & \quad T_{\mathtt{P}_1,X} \cup T_{\mathtt{P}_2,X} \cup (Q^{\mathbf{fin}}_{\mathtt{P}_1,X} \xrightarrow{\epsilon} Q^{\mathbf{init}}_{\mathtt{P}_2,X})\rangle \\ A_{\mathtt{(P_1|P_2)},X} & = & \langle Q_{\mathtt{P}_1,X} \cup Q_{\mathtt{P}_2,X},\ Q^{\mathbf{init}}_{\mathtt{P}_1,X} \cup Q^{\mathbf{init}}_{\mathtt{P}_2,X},\ Q^{\mathbf{fin}}_{\mathtt{P}_1,X} \cup Q^{\mathbf{fin}}_{\mathtt{P}_2,X},\ T_{\mathtt{P}_1,X} \cup T_{\mathtt{P}_2,X}\rangle \\ A_{\mathtt{P*},X} & = & \langle\ Q_{\mathtt{P},X} \cup \{q_1,q_2\},\ n\{q_1\},\ \{q_2\}, \\ & & \quad T_{\mathtt{P},X} \cup ((Q^{\mathbf{fin}}_{\mathtt{P},X} \cup \{q_1\}) \xrightarrow{\epsilon} (Q^{\mathbf{init}}_{\mathtt{P},X} \cup \{q_2\}))\rangle \\ A_{\mathtt{(P\ as\ x)},X} & = & A_{\mathtt{P},X\cup\{\mathtt{x}\}} \end{array}$$

Here, $Q_1 \xrightarrow{l} Q_2$ means $\{q_1 \xrightarrow{l} q_2 \mid q_1 \in Q_1,\ q_2 \in Q_2\}$.

We can prove that the original pattern and the automaton obtained from the translation behave the same.

A.1 Theorem: For any value v, we have $v \in \mathtt{P} \Rightarrow \mathtt{V}$ if and only if $A^{\mathbf{all}}_{\mathtt{P},\emptyset} \vdash v \Rightarrow \rho$ with $\mathbf{env}(\rho)(x) = \mathtt{V}(x)$ for each x.

Proof: We show this theorem by proving a stronger statement:

> For any value v, we have $v \in \mathtt{P} \Rightarrow \mathtt{V}$ if and only if $A^{\mathbf{all}}_{\mathtt{P},X} \vdash v \Rightarrow \rho$ with $\mathbf{env}(\rho)(x) = \mathtt{V}(x)$ for each $x \in \mathbf{bv}(\mathtt{P}) \setminus X$.

Induction on the lexicographic order of $(|v|, |\mathtt{P}|)$. We below show only interesting cases; other cases follows from a straightforward use of the induction hypothesis.

Case $\mathtt{P}=\mathtt{P}_1,\mathtt{P}_2$:

By the induction hypothesis, we have $v \in \mathtt{P}_i \Rightarrow \mathtt{V}$ if and only if $A^{\mathbf{all}}_{\mathtt{P}_i,X} \vdash v \Rightarrow \rho$ with $\mathbf{env}(\rho)(x) = \mathtt{V}(x)$ for each $x \in \mathbf{bv}(\mathtt{P}_i) \setminus X$ for $i = 1, 2$. The remaining is to show: $A^{\mathbf{all}}_{\mathtt{P}_1,X} \vdash v_1 \in q \Rightarrow \rho_1$ and $A^{\mathbf{all}}_{\mathtt{P}_2,X} \vdash v_2 \Rightarrow \rho_2$ if and only if $A^{\mathbf{all}}_{(\mathtt{P}_1,\mathtt{P}_2),X} \vdash v \in q \Rightarrow \rho$ with $q \in Q_{\mathtt{P}_1,X}$, $v = v_1, v_2$, and $\rho = \rho_1, \rho_2$. We can prove the "only if" direction by induction on the derivation of $A^{\mathbf{all}}_{\mathtt{P}_1,X} \vdash v_1 \in q \Rightarrow \rho_1$, and the converse by induction on the derivation of $A^{\mathbf{all}}_{(\mathtt{P}_1,\mathtt{P}_2),X} \vdash v \in q \Rightarrow \rho$.

Case $\mathtt{P}=\mathtt{P}'\mathtt{*}$:

By the induction hypothesis, we have $v \in \mathtt{P}' \Rightarrow \mathtt{V}$ if and only if $A^{\mathbf{all}}_{\mathtt{P}',X} \vdash v \Rightarrow \rho$ with $\mathbf{env}(\rho)(x) = \mathtt{V}(x)$ for each $x \in \mathbf{bv}(\mathtt{P}') \setminus X$. The remaining is to show: $A^{\mathbf{all}}_{\mathtt{P}',X} \vdash v_0 \in q \Rightarrow \rho_0$ and $A^{\mathbf{all}}_{\mathtt{P}',X} \vdash v_i \Rightarrow \rho_i$ for $i = 1, \ldots, n$ if and only if $A^{\mathbf{all}}_{\mathtt{P}'\mathtt{*},X} \vdash v \in q \Rightarrow \rho$ with $q \in Q^{\mathbf{all}}_{\mathtt{P}',X}$, $v = v_0, v_1, \ldots, v_n$ and $\rho = \rho_0, \rho_1, \ldots, \rho_n$. We can prove the "only if" direction by induction on the lexicographic order of the pair of the derivation of $A^{\mathbf{all}}_{\mathtt{P}',X} \vdash v_0 \in q \Rightarrow \rho_0$ and n, and the converse by induction on the derivation of $A^{\mathbf{all}}_{\mathtt{P}'\mathtt{*},X} \vdash v \in q \Rightarrow \rho$, where n coincides with the number of transitions from $Q^{\mathbf{fin}} \xrightarrow{\epsilon} (Q^{\mathbf{init}} \cup \{q_2\})$ to be used in matching v.

Case $\mathtt{P}=(\mathtt{P}'\ \mathtt{as}\ \mathtt{x})$:

We can show that, for any $\mathtt{P}$ and X, if $A_{\mathtt{P},X} \vdash v \Rightarrow \rho$ and $x \in X$, then $\mathbf{env}(\rho)(x) = v$, by induction on the structure of $\mathtt{P}$. The result follows from the induction hypothesis. ∎

Next, we prove that the ambiguity of a given pattern is preserved by the translation. First, the following corollary of the last theorem is useful in the proof.

A.2 Corollary: $v \in \mathtt{P}$ if and only if $A^{\mathbf{all}}_{\mathtt{P},X} \vdash v \Rightarrow \pi$ for some π

Proof: The result follows from Theorem A.1 and the easy fact that $A^{\mathbf{all}}_{\mathtt{P},X} \vdash v \Rightarrow \rho$ for some ρ if and only if $A^{\mathbf{all}}_{\mathtt{P},X} \vdash v \Rightarrow \pi$ for some π. ∎

A.3 Theorem: $v \in^a \mathtt{P}$ if and only if $A^{\mathbf{all}}_{\mathtt{P},X}$ is ambiguous for v.

Proof: The "only if" direction proceeds by induction on the derivation of $\mathtt{v} \in^a \mathtt{P}$.

Case AMB-AS: $\mathtt{P} = \mathtt{P}_1\ \mathtt{as}\ \mathtt{x}$ $\quad \mathtt{v} \in^a \mathtt{P}_1$

By the induction hypothesis, $A^{\mathbf{all}}_{\mathtt{P}_1,X\cup\{x\}} \vdash v \Rightarrow \pi_1$ and $A^{\mathbf{all}}_{\mathtt{P}_1,X\cup\{x\}} \vdash v \Rightarrow \pi_1'$ with $\pi_1 \neq \pi_1'$. The result follows since $A^{\mathbf{all}}_{\mathtt{P}_1,X\cup\{x\}} = A^{\mathbf{all}}_{\mathtt{P},X}$ by definition.

Case AMB-LAB: $\mathtt{P} = \mathtt{L}\mathtt{[X]}$ $\quad \mathtt{v} = \mathtt{l}\mathtt{[}\mathtt{v}_1\mathtt{]}$ $\quad \mathtt{l} \in \mathtt{L}$ $\quad \mathtt{v}_1 \in^a E(\mathtt{X})$

By the induction hypothesis, $A^{\mathbf{all}}_{E(\mathtt{X}),\emptyset} \vdash v_1 \Rightarrow \pi_1$ and $A^{\mathbf{all}}_{E(\mathtt{X}),\emptyset} \vdash v_1 \Rightarrow \pi_1'$ with $\pi_1 \neq \pi_1'$. Let q_1, q_2 be the states introduced in constructing $A_{\mathtt{P},X}$. Then, $A^{\mathbf{all}}_{\mathtt{P},X} \vdash v \Rightarrow q_1[\pi_1], q_2$ and $A^{\mathbf{all}}_{\mathtt{P},X} \vdash v \Rightarrow q_1[\pi_1'], q_2$. The result follows.

Case AMB-CAT1: $\mathtt{v} = \mathtt{v}_1\,\mathtt{,}\,\mathtt{v}_2 = \mathtt{v}_1\mathtt{'}\,\mathtt{,}\,\mathtt{v}_2\mathtt{'} \quad \mathtt{v}_1 \neq \mathtt{v}_1\mathtt{'}$
$\mathtt{v}_1 \in \mathtt{P}_1 \quad \mathtt{v}_2 \in \mathtt{P}_2 \quad \mathtt{v}_1\mathtt{'} \in \mathtt{P}_1 \quad \mathtt{v}_2\mathtt{'} \in \mathtt{P}_2$

By Corollary A.2, we have $A^{\mathbf{all}}_{\mathtt{P}_i,X} \vdash v_i \Rightarrow \pi_i$ and $A^{\mathbf{all}}_{\mathtt{P}_i,X} \vdash v_i' \Rightarrow \pi_i'$ for $i = 1, 2$. By definition of A, we have $A^{\mathbf{all}}_{\mathtt{P},X} \vdash v \Rightarrow \pi_1, \pi_2$ and $A^{\mathbf{all}}_{\mathtt{P},X} \vdash v \Rightarrow \pi_1', \pi_2'$. Since $v_1 \neq v_1'$, we have $\pi_1 \neq \pi_1'$. Further, since $\mathbf{st}(A_{\mathtt{P}_1,X}) \cap \mathbf{st}(A_{\mathtt{P}_2,X}) = \emptyset$, we have $\pi_1, \pi_2 \neq \pi_1', \pi_2'$. The result follows.

Case AMB-CAT2: $\mathtt{P} = \mathtt{P}_1\,\mathtt{,}\,\mathtt{P}_2 \quad \mathtt{v} = \mathtt{v}_1\,\mathtt{,}\,\mathtt{v}_2$
$\mathtt{v}_1 \in^a \mathtt{P}_1 \quad \mathtt{v}_2 \in \mathtt{P}_2$

By the induction hypothesis, $A^{\mathbf{all}}_{\mathtt{P}_1,X} \vdash v_1 \Rightarrow \pi_1$ and $A^{\mathbf{all}}_{\mathtt{P}_1,X} \vdash v_1 \Rightarrow \pi_1'$ with $\pi_1 \neq \pi_1'$. Moreover, by Corollary A.2, we have $A^{\mathbf{all}}_{\mathtt{P}_2,X} \vdash v_2 \Rightarrow \pi_2$. By the definition of A, we obtain $A^{\mathbf{all}}_{\mathtt{P},X} \vdash v \Rightarrow \pi_1, \pi_2$ and $A^{\mathbf{all}}_{\mathtt{P},X} \vdash v \Rightarrow \pi_1', \pi_2$. The result follows.

Case AMB-CAT3: $\mathtt{P} = \mathtt{P}_1\,\mathtt{,}\,\mathtt{P}_2 \quad \mathtt{v} = \mathtt{v}_1\,\mathtt{,}\,\mathtt{v}_2$
$\mathtt{v}_1 \in \mathtt{P}_1 \quad \mathtt{v}_2 \in^a \mathtt{P}_2$

Similar to the previous case.

Case AMB-OR1: $\mathtt{P} = \mathtt{P}_1\,\mathtt{|}\,\mathtt{P}_2 \quad \mathtt{v} \in \mathtt{P}_1 \quad \mathtt{v} \in \mathtt{P}_2$

The result follows from Corollary A.2 and $\mathbf{st}(A_{\mathtt{P}_1,X}) \cap \mathbf{st}(A_{\mathtt{P}_2,X}) = \emptyset$.

Case AMB-OR2/OR3:

The result follows from straightforward uses of the induction hypothesis.

Case AMB-REP1: $\mathtt{P} = \mathtt{P}_1\mathtt{*} \quad \mathtt{v} = \mathtt{v}_1\mathtt{,..,}\mathtt{v}_n \quad \mathtt{v} = \mathtt{v}_1\mathtt{'}\mathtt{,..,}\mathtt{v}_m\mathtt{'}$
$(\mathtt{v}_1, \ldots, \mathtt{v}_n) \neq (\mathtt{v}_1\mathtt{'}, \ldots, \mathtt{v}_m\mathtt{'})$
$\mathtt{v}_i \in \mathtt{P}_1$ for each i $\quad \mathtt{v}_i\mathtt{'} \in \mathtt{P}_1$ for each i

By Corollary A.2, we have $A^{\mathbf{all}}_{\mathtt{P}_1,X} \vdash v_i \Rightarrow \pi_i$ for $i = 1, \ldots, n$ and $A^{\mathbf{all}}_{\mathtt{P}_1,X} \vdash v_i \Rightarrow \pi_i'$ for $i = 1, \ldots, m$. By definition of A, we have $A^{\mathbf{all}}_{\mathtt{P},X} \vdash v \Rightarrow \pi_1, \ldots, \pi_n$ and $A^{\mathbf{all}}_{\mathtt{P},X} \vdash v \Rightarrow \pi_1', \ldots, \pi_m'$. Since $(\mathtt{v}_1, \ldots, \mathtt{v}_n) \neq (\mathtt{v}_1\mathtt{'}, \ldots, \mathtt{v}_m\mathtt{'})$, we have $(\pi_1, \ldots, \pi_n) \neq (\pi_1', \ldots, \pi_m')$. Note that, from the way that automata are constructed, each π_i or π_i' ends with a state from $\mathbf{fin}(A_{\mathtt{P},X})$ and there is no transition from this state in $A_{\mathtt{P},X}$. Therefore $\pi_1, \ldots, \pi_n \neq \pi_1', \ldots, \pi_m'$. The result follows.

Case AMB-REP2: $\mathtt{P} = \mathtt{P}_1\mathtt{*} \quad \mathtt{v} = \mathtt{v}_1\mathtt{,..,}\mathtt{v}_n$
$\mathtt{v}_i \in \mathtt{P}_1$ for each i $\quad \mathtt{v}_j \in^a \mathtt{P}_1$ for some j

By Corollary A.2, we have $A^{\mathbf{all}}_{\mathtt{P}_1,X} \vdash v_i \Rightarrow \pi_i$ for each i. By the induction hypothesis, $A^{\mathbf{all}}_{\mathtt{P}_1,X} \vdash v_j \Rightarrow \pi_j'$ and $A^{\mathbf{all}}_{\mathtt{P}_1,X} \vdash v_j \Rightarrow \pi_j''$ with $\pi_j' \neq \pi_j''$. Therefore

$$A^{\mathbf{all}}_{\mathtt{P},X} \vdash v \Rightarrow \pi_1, \ldots, \pi_{j-1}, \pi_j', \pi_{j+1}, \ldots, \pi_n$$

and

$$A^{\mathbf{all}}_{\mathtt{P},X} \vdash v \Rightarrow \pi_1, \ldots, \pi_{j-1}, \pi_j'', \pi_{j+1}, \ldots, \pi_n.$$

The result follows.

The "if" direction proceeds by induction on the lexicographic order of $(|\mathtt{P}|, |v|)$.

Case: $\mathtt{P} = \mathtt{(P_1\ as\ x)}$ or $\mathtt{P} = \mathtt{L[X]}$

The result follows from a straightforward use of the induction hypothesis.

Case: $\mathtt{P} = \mathtt{()}$

This case never arises since always $v = \epsilon$ and $A^{\mathbf{all}}_{\mathtt{P},X} \vdash v \Rightarrow \epsilon$.

Case: $\mathtt{P} = \mathtt{P}_1\,\mathtt{,}\,\mathtt{P}_2$

By the definition of A, we have $A^{\mathbf{all}}_{\mathtt{P}_i,X} \vdash v_i \Rightarrow \pi_i$ and $A^{\mathbf{all}}_{\mathtt{P}_i,X} \vdash v'_i \Rightarrow \pi'_i$ for $i = 1, 2$ where $v = v_1, v_2 = v'_1, v'_2$ with $\pi = \pi_1, \pi_2$ and $\pi' = \pi'_1, \pi'_2$. We have two cases.

- $v_1 = v'_1$ and $v_2 = v'_2$. From $\pi \neq \pi'$, either $\pi_1 \neq \pi'_1$ or $\pi_2 \neq \pi'_2$. The result follows from the induction hypothesis and AMB-Cat2, in the first case, or AMB-Cat3 in the second case.
- $v_1 \neq v'_1$ and $v_2 \neq v'_2$. The result follows from Corollary A.2 and AMB-Cat1.

Case: $\mathtt{P} = \mathtt{P}_1 \,|\, \mathtt{P}_2$

By the definition of A, we have four cases.

- $A^{\mathbf{all}}_{\mathtt{P}_1,X} \vdash v \Rightarrow \pi$ and $A^{\mathbf{all}}_{\mathtt{P}_1,X} \vdash v \Rightarrow \pi'$. The result follows from the induction hypothesis and AMB-Or2.
- $A^{\mathbf{all}}_{\mathtt{P}_2,X} \vdash v \Rightarrow \pi$ and $A^{\mathbf{all}}_{\mathtt{P}_2,X} \vdash v \Rightarrow \pi'$. The result follows from the induction hypothesis and AMB-Or3.
- $A^{\mathbf{all}}_{\mathtt{P}_1,X} \vdash v \Rightarrow \pi$ and $A^{\mathbf{all}}_{\mathtt{P}_2,X} \vdash v \Rightarrow \pi'$, or $A^{\mathbf{all}}_{\mathtt{P}_2,X} \vdash v \Rightarrow \pi$ and $A^{\mathbf{all}}_{\mathtt{P}_1,X} \vdash v \Rightarrow \pi'$. The result follows from Corollary A.2 and AMB-Or1.

Case: $\mathtt{P} = \mathtt{P}_1\mathtt{*}$

By the definition of A, we have $A^{\mathbf{all}}_{\mathtt{P}_i,X} \vdash v_i \Rightarrow \pi_i$ for $i = 1, \ldots, n$ and $A^{\mathbf{all}}_{\mathtt{P}_i,X} \vdash v'_i \Rightarrow \pi'_i$ for $i = 1, \ldots, m$ where $v = v_1, \ldots, v_n = v'_1, \ldots, v'_m$ with $\pi = \pi_1, \ldots, \pi_n$ and $\pi' = \pi'_1, \ldots, \pi'_m$. We have two cases.

- $(v_1, \ldots, v_n) = (v'_1, \ldots, v'_m)$. From $\pi \neq \pi'$, we have $\pi_i \neq \pi'_i$ for some i. The result follows from the induction hypothesis and AMB-Rep2.
- $(v_1, \ldots, v_n) \neq (v'_1, \ldots, v'_m)$. The result follows from Corollary A.2 and AMB-Rep1. ∎

B First-match semantics

In this section, we describe the first-match semantics of patterns. The technical materials here are mostly quoted from [HP02].

For the treatment of the first-match policy, we use a notion of *choice sequence*. During pattern matching, we remember the index of the branch we take at each choice point. A choice sequence is a sequence of such indices, listed according to the order of traversal—from left to right and from outer to inner. Finally, we take the *smallest* choice sequence in the dictionary order.

Formally, we describe the semantics of patterns by first defining the relation $\mathtt{v} \in \mathtt{P} \Rightarrow \mathtt{V} \;/\; \alpha$, read "v is matched by P, yielding V and α," where a choice sequence α is a sequence of elements from the ordered set $(\{1, 2\}, \leq)$

and an environment $\mathtt{V}$ is a finite mapping from variables to values (written $\mathtt{x}_1\mathtt{:v}_1\mathtt{,..,x}_n\mathtt{:v}_n$). Then we define the relation $\mathtt{v} \in \mathtt{P} \Rightarrow \mathtt{V}$ that takes the smallest choice sequence w.r.t. the lexicographic order $\sqsubset$ on choice sequences. The following set of rules defines both relations.

$$\frac{\mathtt{v} \in \mathtt{P} \Rightarrow \mathtt{V} \;/\; \alpha}{\mathtt{v} \in \mathtt{(P\ as\ x)} \Rightarrow \mathtt{x:v,V} \;/\; \alpha} \qquad \text{(EP-As)}$$

$$\mathtt{()} \in \mathtt{()} \Rightarrow \emptyset \;/\; \cdot \qquad \text{(EP-Emp)}$$

$$\frac{\mathtt{l} \in \mathtt{L} \qquad \mathtt{v} \in E(\mathtt{X}) \Rightarrow \mathtt{V} \;/\; \alpha}{\mathtt{l[v]} \in \mathtt{L[X]} \Rightarrow \mathtt{V} \;/\; \alpha} \qquad \text{(EP-Lab)}$$

$$\frac{\mathtt{v} \in \mathtt{P} \Rightarrow \mathtt{V} \;/\; \alpha \qquad \mathtt{w} \in \mathtt{Q} \Rightarrow \mathtt{W} \;/\; \beta}{\mathtt{v,w} \in \mathtt{P,Q} \Rightarrow \mathtt{V,W} \;/\; \alpha, \beta} \qquad \text{(EP-Cat)}$$

$$\frac{\mathtt{v} \in \mathtt{P} \Rightarrow \mathtt{V} \;/\; \alpha}{\mathtt{v} \in \mathtt{P|Q} \Rightarrow \mathtt{V} \;/\; 1, \alpha} \qquad \text{(EP-Or1)}$$

$$\frac{\mathtt{v} \in \mathtt{Q} \Rightarrow \mathtt{V} \;/\; \alpha}{\mathtt{v} \in \mathtt{P|Q} \Rightarrow \mathtt{V} \;/\; 2, \alpha} \qquad \text{(EP-Or2)}$$

$$\frac{\begin{array}{c}\mathtt{v} \in \mathtt{P} \Rightarrow \mathtt{V} \;/\; \alpha \\ \forall \beta.(\mathtt{v} \in \mathtt{P} \Rightarrow \mathtt{U} \;/\; \beta \implies \alpha \sqsubset \beta)\end{array}}{\mathtt{v} \in \mathtt{P} \Rightarrow \mathtt{V}} \qquad \text{(EP)}$$

Notice that in EP-Cat we concatenate the choice sequences left to right, and that in ET-Or1 and ET-Or2 we adjoin the choice number to the front. These reflect the policy that the priority of choice is from left to right and from outer to inner.

In the above definition, the first match may not be definable for some patterns. For example, suppose that we want to match the value `a[]` against the pattern `Y` where:

```
pat Y = Y | a[]
```

Clearly, the second clause `a[]` will match the value. But since the first clause `Y` has higher priority, we should examine this first. Unfolding this pattern name, we see the pattern `(Y|a[])` again and therefore the same argument can be repeated. Thus, for any match of the value against this pattern, we can find another match with higher priority; therefore there is no first match. Notice that this anomaly arises because the unguarded recursive use of `Y` has no pattern in front of it and therefore it recurs without decreasing the size of the input value. In order to ensure that each pattern has the first match for all input values, we impose a syntactic restriction called "no head-recursion," where any unguarded recursive use of pattern names must be preceded by a pattern that does not match the empty sequence.

References

[AM91] Andrew W. Appel and David B. MacQueen. Standard ML of New Jersey. In *Third Int'l Symp. on Prog. Lang. Implementation and Logic Programming*, pages 1–13. Springer-Verlag, August 91.

[AQM+97] Serge Abiteboul, Dallan Quass, Jason McHugh, Jennifer Widom, and Janet L. Wiener. The Lorel query language for semistructured data. *International Journal on Digital Libraries*, 1(1):68–88, 1997.

[BCF02] V. Benzaken, G. Castagna, and A. Frisch. Cduce: a white paper. In *PLAN-X: Programming Language Technologies for XML*, 2002.

[BEGO71] Ronald Book, Shimon Even, Seila Greibach, and Gene Ott. Ambiguity in graphs and expressions. *IEEE Transactions on Computers*, 20:149–153, 1971.

[BK93] Anne Brüggemann-Klein. Regular expressions into finite automata. *Theoretical Computer Science*, 120:197–213, 1993.

[CDG+99] Hubert Comon, Max Dauchet, Rémy Gilleron, Florent Jacquemard, Denis Lugiez, Sophie Tison, and Marc Tommasi. Tree automata techniques and applications. Draft book; available electronically on `http://www.grappa.univ-lille3.fr/tata`, 1999.

[CG00] Luca Cardelli and Giorgio Ghelli. A query language for semistructured data based on the Ambient Logic. Manuscript, April 2000.

[Cla99] James Clark. XSL Transformations (XSLT), 1999. `http://www.w3.org/TR/xslt`.

[CM01] James Clark and Makoto Murata. RELAX NG. `http://www.relaxng.org`, 2001.

[CS98] Sophie Cluet and Jérôme Siméon. Using YAT to build a web server. In *Intl. Workshop on the Web and Databases (WebDB)*, 1998.

[DFF+98] Alin Deutsch, Mary Fernandez, Daniela Florescu, Alon Levy, and Dan Suciu. XML-QL: A Query Language for XML, 1998. `http://www.w3.org/TR/NOTE-xml-ql`.

[FCB02] Alain Frisch, Giuseppe Castagna, and Véronique Benzaken. Semantic subtyping. In *Seventeenth Annual IEEE Symposium on Logic In Computer Science*, 2002.

[HM02] Haruo Hosoya and Makoto Murata. Validation and boolean operations for attribute-element constraints. In *Programming Languages Technologies for XML (PLAN-X)*, pages 1–10, 2002.

[HP02] Haruo Hosoya and Benjamin C. Pierce. Regular expression pattern matching for XML. *Journal of Functional Programming*, 2002. To appear. Short version appeared in Proceedings of The 25th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 67–80, 2001.

[HP03] Haruo Hosoya and Benjamin C. Pierce. XDuce: A typed XML processing language. *ACM Transactions on Internet Technology*, 2003. To appear.

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

[Kaw01] Kohsuke Kawaguchi. Ambiguity detection of RELAX grammars. `http://www.kohsuke.org/relaxng/ambiguity/AmbiguousGrammarDetection.pdf`, 2001.

[LVD+96] Xavier Leroy, Jérôme Vouillon, Damien Doligez, et al. The Objective Caml system. Software and documentation available on the Web, `http://pauillac.inria.fr/ocaml/`, 1996.

[MS99] Tova Milo and Dan Suciu. Type inference for queries on semistructured data. In *Proceedings of Symposium on Principles of Database Systems*, pages 215–226, Philadelphia, May 1999.

[MSV00] Tova Milo, Dan Suciu, and Victor Vianu. Typechecking for XML transformers. In *Proceedings of the Nineteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 11–22. ACM, May 2000.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. The MIT Press, 1990.

[Mur97] Makoto Murata. Transformation of documents and schemas by patterns and contextual conditions. In *Principles of Document Processing '96*, volume 1293 of *Lecture Notes in Computer Science*, pages 153–169. Springer-Verlag, 1997.

[NS98] Andreas Neumann and Helmut Seidl. Locating matches of tree patterns in forests. In *18th FSTTCS*, volume 1530 of *LNCS*, pages 134–145, 1998.

[NS00] Frank Neven and Thomas Schwentick. Expressive and efficient pattern languages for tree-structured data. In *Proceedings of the Nineteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 145–156. ACM, 2000.

[PHH+93] Simon L. Peyton Jones, Cordelia V. Hall, Kevin Hammond, Will Partain, and Philip Wadler. The Glasgow Haskell compiler: a technical overview. In *Proc. UK Joint Framework for Information Technology (JFIT) Technical Conference*, July 1993.

[PV00] Yannis Papakonstantinou and Victor Vianu. DTD Inference for Views of XML Data. In *Proceedings of the Nineteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 35–46, Dallas, Texas, May 2000.

[Toz01] Akihiko Tozawa. Towards static type inference for XSLT. In *Proceedings of ACM Symposium on Document Engineering*, 2001.

[Van03] Stijn Vansummeren. Unique pattern matching in strings. 2003.

Acknowledgments

I am grateful to Benjamin Pierce, Peter Buneman, Jérôme Vouillon, Michael Levin, and Vladimir Gapeyev for useful discussions and comments.