Compiling Regular Patterns to Sequential Machines

Burak Emir

École Polytechnique Fédérale de Lausanne
1015 Lausanne, Switzerland

Technical Report IC/2004/72

1 Introduction

Many programming languages have a pattern matching construct that can be generalized to deal with regular expressions. This is especially useful for decomposing semi-structured data, i.e. labeled, attributed trees where the children of a node form a sequence of *arbitrary* length. Schema languages make use of regular expressions to constrain such sequences in instance documents. This has led to interest on regular pattern matching for semistructured data like in XDUCE [13], CDUCE [1]. For general-purpose programming, regular pattern matching is used in XTATIC [10, 9], HARP [4] (an extension of HASKELL) and SCALA [21]. A related concept are XEN filters [17] and node-selection in the XPATH standard. Moreover, a recent criticism of current schema languages admonishes the lack of "user-defined simple datatypes", where a data type is seen as a regular expression with variables. Such a type constrains the text content of elements and, at the same time, gives names to parts of the content (imagine a datatype for date, with access to month,year). Another example for regular patterns are POSIX regular expressions (used in EMACS-LISP, PERL) which yield the matches of parenthesized subexpressions. In most compilers, a lexical analyzer matches parts of a source file against regular expressions, applying the longest match rule. To these long-known applications have been added new ones, like finding segments in DNA sequences or content-based routing and deep packet classification.

Regular patterns are a natural generalization of pattern matching as known from ML and HASKELL. For the sake of an example, let us consider matching sequences of characters, with the convention that _ is a wildcard pattern and a binding pattern $v@p$ matches everything p matches, binding the result to the variable v. An email can be succinctly queried for its sender:

$$(\text{`F`},\ \text{`r`},\ \text{`o`},\ \text{`m`},\ \text{`:`},\ x@\_^\star,\ \text{`\n`},\ \_^\star)$$

We are interested in everything between "From:" and the first newline character '\n'. But the binding pattern $x@\_^\star$ might possibly stretch far beyond the first newline, because the patterns $\_^\star$ match arbitrarily many arbitrary characters. The pattern is ambiguous; several values for x are possible.

Ambuiguity can be removed by imposing a match policy. The preference to stop binding at the *first* newline character means that we are interested in the *shortest* match. Clearly, variables can make regular expression patterns ambiguous also in the absence of wildcards, as in the minimal example $(x@\text{`a`}^\star,\ y@\text{`a`}^\star)$.

In this paper, we will give a formal model of the shortest match and derive our algorithms from it. This model is based on the metaphor that binding to a variable can be seen as tagging parts of the input with the variable to which they are bound. Let us agree on writing ${}_x a$ for appending input element a to variable x. Then matching "From:jj@foo.net\n..." against the email pattern above to yield the substitution $\{x \mapsto \text{"jj@foo.net"}\}$ is conceptually the same as transforming the input string to

$$\text{"}\ \mathtt{F}\ \mathtt{r}\ \mathtt{o}\ \mathtt{m}\ \mathtt{:}\ {}_x\mathtt{j}\,{}_x\mathtt{j}\,{}_x@\,{}_x\mathtt{f}\,{}_x\mathtt{o}\,{}_x\mathtt{o}\,{}_x\mathtt{.}\,{}_x\mathtt{n}\,{}_x\mathtt{e}\,{}_x\mathtt{t}\backslash\mathtt{n}\ldots\text{"}.$$

The desired substitution can be read off the output. This view does not depend on any sequence representations, and it generalizes well to trees. In particular, it can also serve to reason about all-matches style pattern matching as expressed by XPATH. The shortest match and the semantics of pattern matching can be concisely specified by reasoning on such annotated strings. Since these bindings are annotated copies of input words of a regular language, we obtain a natural formulation of pattern matching as length-preserving transduction of words.

Contributions. The contributions of this paper are (1) the (first) sequential machine formulation of regular pattern matching, (2) giving effective algorithms based on this model which can be used to compile patterns into code that finds the shortest/longest matches in linear time (two traversals of the input), (3) a characterization and decision procedure for cases that need only a single traversal (unambiguous patterns), (4) an intuitive account on match policies and position automata, (5) a discussion on generalization and application of the results to tree matching and XPATH evaluation.

The thoughts discussed in this paper should help implement regular pattern matching, relate different approaches and proof techniques, and prevent implementors from reinventing the wheel.

All presented results are effective. The implementation techniques are used in the reference SCALA compiler.

Related work. While compilation of regular expressions is standard, compilation of regular patterns is not. Ambiguities seem to complicate the task of writing an intelligible specification of regular pattern matching. The problem is first mentioned by Hosoya, Vouillon and Pierce [13]. A recent account on regular tree pattern matching by Levin [15] omits the treatment of ambiguities and variables. Our work

aims to provide this missing piece in order to promote regular patterns in programming languages.

Disambiguation of patterns has been specified rigorously by Tabuchi et al. in their work on λ^{re} [22], where the longest match-policy is used in type inference of patterns. We give an efficient implementation that seems compatible with their specification, although we will focus on shortest match instead of longest.

Frisch and Cardelli [8] consider matching with ambiguities in depth, addressing efficient implementation as well. They obtain results similar to ours, but depart from different assumptions: (1) The match policy under consideration is greedy matching, a local approximation of the longest match policy. This yields different bindings, because the order of branches in an alternative pattern $p_1|p_2$ affects the outcome. Our longest/shortest match algorithms have the same runtime complexity and thus complement theirs. We thus refute the claim that greedy matching is easier to implement. (2) Their approach is "expression-centric" - regular expressions denote sets of structured values, which prohibits their rewriting and distinguishes expressions denoting the same language. Our approach is more semantic: a regular expression denotes not more and not less than a regular language. In our approach, rewriting the regular expression turns out to be **indispensable** for longest/shortest match. Apart from that, the semantic choice is arguably better for compilation of pattern matching. It is an advantage to forget the structure of patterns in an early compilation phase, because all interesting decision procedures on regular languages are based on automata.

Broberg, Farre and Svenningsson [4] add a form of regular patterns to Haskell, combining several approaches to handle ambiguities: they use greedy and nongreedy operators, and return all-matches in all other cases. Their compilation scheme is a proof of concept of their language design, and not efficient. The choice of disambiguated operators is also made in the mainstream languages Perl and Java. We show how our approach can be extended to disambiguated operators without sacrificing efficiency.

The two first approaches mention a two-pass construction similar to the one presented here. Also Neumann and Seidl[20] use a two-pass construction to extract matches of *context patterns*, which can be described as a context-aware generalization of XPath.

By connecting the problem to long-standing theoretic evidence, some explanation is given why this kind of construction seems inevitable. We formulate pattern matching as a sequential rational function, and the two-pass construction is a consequence of a long-standing decomposition theorem Elgot and Mezei [6]. For details, we refer the interested reader to the chapter on rational functions in the standard work on transductions [3].

Organization of the paper After a brief preliminary section, we introduce patterns and bindings, which are sequences over a special alphabet, in Section 2. We then introduce the position automata construction in Section 3 and show how the shortest/longest match corresponds to a maximal/minimal run of a position automaton. In Section 4, we introduce sequential machines, and a first naive compilation scheme for pattern matching. In Section 5, we turn to efficient shortest-match pattern matching, using two deterministic sequential machines. We discuss generalizations and applications in Section 6, followed by conclusion, acknowledgement, references and an appendix with proofs.

Preliminary. The empty set is denoted $\emptyset$. For sets A, B we write 2^A for the powerset of A and $A - B$ for set difference. The nonnegative integers $\mathbb{N} = \{1, 2, \ldots\}$ are ordered by $<$, whose reflexive closure is $\leq$. When describing sets through their elements $\{x_1, \ldots, x_k\}$, we always assume the x_i are pairwise distinct and if an order exists, the indices respect it. Σ and Θ are nonempty, finite sets of symbols called alphabets. The domain of a partial function f is denoted $\mathsf{dom}(f)$. Composition is written $f \circ g$, where f is applied first. $S^\star$ is the set of all sequences of symbols from S, ϵ is the empty sequence, and $S^+ = S^\star - \{\epsilon\}$. More precisely, a sequence $w \in S^\star$ is a partial mapping $w : \mathbb{N} \to S$ where $\mathsf{dom}(w)$ is closed under predecessor. Concatenation of sequences u, v is expressed by juxtaposition, as in uv. Sometimes concatenation is written explicitly using the infix $\cdot$ operator. For sets of words, $A \cdot B = \{uw | u \in A, w \in B\}$. For a word w, its reverse is denoted w^{rev}. For set A, we set $A^0 = \{\epsilon\}, A^{i+1} = A \cdot A^i$. The prefix ordering on $\mathbb{N}^\star$ is written $\preceq$, i.e. $u \preceq v$ iff $uz = v$ for some $z \in \mathbb{N}^\star$. For $\mathbb{N}^\star$, we can define the quasi-lexicographical ordering $u <^\star v$ iff $u = wiz$ and $v = wjz'$ for some $i, j \in \mathbb{N}$ and $w, z, z' \in \mathbb{N}^\star$. We write $u <^\star_{\mathsf{right}} v$ if $u^{\mathsf{rev}} <^\star v^{\mathsf{rev}}$. Regarding sequences of numbers as positions (or paths), a tree is a partial mapping $t : \mathbb{N}^\star \to \Sigma$ from positions to symbols where $\mathsf{dom}(t)$ is closed under prefix and left sibling. $<^\star$ is the usual pre-order on tree nodes. These trees are unranked, and nodes are elements of $\mathbb{N}^\star$. ϵ denotes the root node and for a node u, ui is the i-th child of u. The set of all trees is denoted T_Σ. Positions that are maximal w.r.t. $\preceq$ are called leaves, and the set of leaves of t is $\mathsf{frontier}(t)$. Thus for every $w \in \mathsf{frontier}(t)$ we have $w \preceq w'$ implies $w = w'$ for any $w' \in \mathsf{dom}(t)$. When we write $\mathsf{frontier}(t) = \{w_1, \ldots, w_n\}$ we always respect the depth-first, left-to-right ordering of leaves, i.e. $w_i <^\star w_j$ for $i < j$. A tree is written as $t(w)$ if w is a leaf position or as $t(w)[\ldots]$ where the children are written juxtaposed between the brackets. We furthermore assume a countably infinite set of variables V.

2 Regular Patterns

2.1 Pattern Syntax and Denotation

The set $\mathsf{RegExp}(\Sigma)$ of regular expressions over Σ, and the set $\mathsf{RegPat}(\Sigma, V)$ for regular patterns over alphabet Σ and variables V is given in Figure 1. To save some parentheses, operator precedence is *, $\cdot$, $|$. We treat parts that are not bound to any variable as if they were bound to a fresh one. A pattern is a non-empty sequence $x_1@r_1 \cdots x_k@r_k$ of binding patterns $x_i@r_i$, where $x_1, \ldots, x_k$ are distinct variables from V, and $r_1, \ldots, r_k \in \mathsf{RegExp}(\Sigma)$. The set $\{x_1, \ldots, x_k\}$ is denoted $\mathsf{var}(p)$ and is ordered in the obvious way. If not otherwise mentioned, we always talk about a fixed pattern $p = x_1@r_1 \cdots x_k@r_k$.

The definitions are chosen to make patterns denote a regular language on the alphabet $V \times \Sigma$. In fact, for any given pattern p, the alphabet is the finite subset $\mathsf{var}(p) \times \Sigma$. We arrived at this choice following these two basic insights.

The first is that it is helpful to choose *bindings* to be sequences $s \in (V \times \Sigma)^\star$. We write ${}_x a$ for elements from $V \times \Sigma$ to lighten notation throughout the paper. The formal definition of the projection $\mathsf{proj} : V \times \Sigma \to \Sigma$ is omitted, and its extension to sequences and to regular expressions will be used without fanfare. A binding can be viewed as a usual substition via the mapping described in Figure 2.

$$
\begin{array}{lcll}
\mathsf{r} & ::= & \epsilon & [\![\epsilon]\!] = \{\epsilon\} \\
 & & a & [\![a]\!] = \{a\} \qquad (a \in \Sigma) \\
 & & \mathsf{r}_1 \cdot \mathsf{r}_2 & [\![\mathsf{r}_1 \cdot \mathsf{r}_2]\!] = [\![\mathsf{r}_1]\!] \cdot [\![\mathsf{r}_2]\!] \\
 & & \mathsf{r}_1 \mid \mathsf{r}_2 & [\![\mathsf{r}_1 \mid \mathsf{r}_2]\!] = [\![\mathsf{r}_1]\!] \cup [\![\mathsf{r}_n]\!] \\
 & & \mathsf{r}^\star & [\![\mathsf{r}^\star]\!] = \bigcup_{\{i \geq 0\}} [\![\mathsf{r}]\!]^i \\
 & & & \\
\mathsf{p} & ::= & v@\mathsf{r} & [\![v@\mathsf{r}]\!] = [\![\mathbf{vp}(v, \mathsf{r})]\!] \\
 & & \mathsf{p} \cdot \mathsf{p} & [\![\mathsf{p}_1 \cdot \mathsf{p}_2]\!] = [\![\mathsf{p}_1]\!] \cdot [\![\mathsf{p}_2]\!]
\end{array}
$$

Figure 1: Regular expressions, regular patterns, and their denotation

$$
\begin{array}{ll}
\mathbf{bind}(\epsilon) & = \{x \mapsto \epsilon \mid x \in V\} \\
\mathbf{bind}(b' \cdot {}_x a) & = \mathbf{bind}(b') \oplus \{x \mapsto a\}
\end{array}
$$

$$
\{x_i \mapsto w_i\}_{x_i \in V} \oplus \{x_j \mapsto a\} = \{x_j \mapsto w_j \cdot a\} \cup \{x_i \mapsto w_i\}_{x_i \in V, i \neq j}
$$

Figure 2: Definition of **bind**

$$
\begin{array}{ll}
\mathbf{vp}(v, \epsilon) & = \epsilon \\
\mathbf{vp}(v, a) & = {}_v a \qquad (a \in \Sigma) \\
\mathbf{vp}(v, r_1 \cdot r_2) & = \mathbf{vp}(v, r_1) \cdot \mathbf{vp}(v, r_1) \\
\mathbf{vp}(v, r_1 \mid r_2) & = \mathbf{vp}(v, r_2) \mid \mathbf{vp}(v, r_2) \\
\mathbf{vp}(v, r^\star) & = \mathbf{vp}(v, r)^\star
\end{array}
$$

Figure 3: Definition of **vp**

Each element ${}_x a$ contributes to the value of x, in order of appearance from left to right. For instance,

$$
\mathbf{bind}({}_x a_x b_y c_z a) = \{x \mapsto ab,\ y \mapsto c,\ z \mapsto a\}
$$

It is easy to see that $s \in [\![p]\!]$ implies that we can decompose s into $s_1 \cdots s_k$ with $s_i \in (\{x_i\} \times \Sigma)^\star$, and that for each of these pieces it holds that $\mathsf{proj}(s_i) = \mathbf{bind}(b)(x_i)$. Sometimes, the piece s_i is called "binding for variable x_i".

The second insight connects patterns and (regular) sets of bindings. This happens syntactically by pushing down the variables to the leaves, and changing the alphabet via the mapping in Figure 3. Also this mapping is straightforwardly extended to patterns. For instance,

$$
\mathbf{vp}(x@(ab|b^\star)y@(c|\epsilon)z@a^\star) = ({}_x a_x b|{}_x b^\star) \cdot ({}_y c|\epsilon) \cdot {}_z a
$$

2.2 Semantics of Matching

Pattern matching with variable binding consists of recognizing whether a word $w \in \Sigma^\star$ matches, and if it does, in providing a suitable binding $s \in (V \times \Sigma)^\star$ for the variables. For a pattern $p = x_1@r_1 \cdots x_k@r_k$, it must hold that $w \in r_1 \cdots r_k$, and that a binding s be produced with $\mathbf{bind}(s)(x_i) \in [\![r_i]\!]$ for each i. Since bindings are annotated input words, the problem consists of transforming words $w \in \Sigma^\star$ into bindings $s \in (V \times \Sigma)^\star$. The transformation is very particular, since it merely annotates input words. It is thus sufficient to specify matching as generation of a suitable binding.

We will see later, that we can turn automata transitions ${}_x a$ into sequential machine transition $a : {}_x a$, i.e. one that reads a and writes ${}_x a$ at runtime. We should mention here that we use sequential machines as conceptual devices. Actual generated sequential machine code does not write bindings; instead the environment is modified directly (e.g. by modifying a pointer-structure on the heap).

A word $w \in \Sigma^\star$, a pattern $p \in \mathsf{RegPat}(\Sigma, V)$ and a binding $s \in (V \times \Sigma)^\star$ are in the ternary matching relation $w \triangleright p \Rightarrow s$ (pronounced "w matches p yielding s") if

1. $\mathsf{proj}(s) = w$ and
2. $s \in [\![p]\!]$.

Proposition 1
If $w \triangleright p \Rightarrow s_1 \cdots s_k$ then $w \in [\![\mathsf{proj}(p)]\!]$ and $\mathsf{proj}(s_i) \in r_i$ for all $x_i \in \mathsf{var}(p)$.

We denote by $\mathsf{Env}(p) \subseteq \Sigma^\star \times (V \times \Sigma)^\star$ the relation on words and bindings induced by p. We also write $\mathsf{Env}(p, w) = \{\ s \mid w \triangleright p \Rightarrow s\ \}$ for the set of possible bindings for pattern p and word w.

Let us consider some examples. The word abb matches the regular pattern $x@a^\star y@b^\star$, yielding ${}_x a_y b_y b$. For a^m, the pattern $x@a^\star y@a^\star$ yields $m + 1$ possible bindings. For the word $w = aaabbb$ and $p = x@a^\star y@a(ab)^\star z@b^\star$ we have $\mathsf{Env}(p, w) = \{s, s'\}$ with $\mathbf{bind}(s) = \{x \leftarrow a, y \leftarrow aab, z \leftarrow bb\}$ and $\mathbf{bind}(s') = \{x \leftarrow aa, y \leftarrow a, z \leftarrow bbb\}$.

The latter two examples are ambiguous patterns, which for a matching word yield more than one binding. In such a case, a match policy determines which of the possibly several bindings should be generated.

2.3 Specification of Shortest Match

The shortest (or right-longest) match policy means that, starting from the right, we need to choose the longest *possible* binding for each variable of a pattern. In the example $x@a^\star y@a^\star$, this would mean to always choose $\{y \leftarrow w\}$. The longest (or left-longest) match policy is defined symmetrically. The use of the word "possible" here underlines that the rest of the word has to match the rest of the pattern. This can require backtracking. Consider for instance $w \triangleright p \Rightarrow s'$ as in the last example:

$$
\overbrace{a}^{a^\star} \cdot \overbrace{a}^{a^\star} \cdot \overbrace{a}^{a^\star} \cdot \overbrace{b}^{a(ab)^\star} \cdot \overbrace{b \cdot b \cdot b}^{b^\star}
$$

The indicated match is at the same time left-longest and right-longest. Consuming all a's to match $x@a^\star$ does not yield any match, let along the longest; binding the first a to y will fail to construct the shortest. We will focus on the shortest (or right-longest) match from now on. There is a total order on $\mathsf{Env}(p, w)$, which tells whether a binding is right-longer. Let $s = s_1 \cdots s_k$ and $s' = s'_1 \cdots s'_k$. Then we define $s >_{\mathsf{right}} s'$ if either

- $k = 1$ and $|s| > |s'|$, or

- $|s_k| > |s'_k|$, or
- $|s_k| = |s'_k|$ and $s_1 \cdots s_{k-1} >_{\mathsf{right}} s'_1 \cdots s'_{k-1}$

In other words, $>_{\mathsf{right}}$ is the lexicographical order on the lengths of the s_i, with k being the most significant position. The definition of $>_{\mathsf{left}}$ is the other way round, making 1 the most significant position.

The right-longest match is the maximal element w.r.t. $>_{\mathsf{right}}$. It is easy to see that it exists and is unique, because the order is total.

2.4 Minimal length

We would like to have a correspondence between (leaf) positions in expressions and states in automata, in order to argue about the shortest match. But this is hindered by the fact that in branches of an alternation, it is not their order of appearance in the expression that counts, but their *minimal length*. It is a technical property defined as

$$\begin{array}{ll} \mathsf{minlen}(\epsilon) & = 0 \\ \mathsf{minlen}(a) & = 1 \\ \mathsf{minlen}(r_1 \cdot r_2) & = \mathsf{minlen}(r_1) + \mathsf{minlen}(r_2) \\ \mathsf{minlen}(r_1|r_2) & = \min\{\mathsf{minlen}(r_1), \mathsf{minlen}(r_2)\} \\ \mathsf{minlen}(r^\star) & = 0 \end{array}$$

Alternation is commutative, so our semantics allows us to rewrite regular expressions in a way that establishes our desired correspondence between minimal length and positions. For right-longest match, we can rewrite a regular expression such that for every alternation $r_1|\ldots|r_m$ we have $\mathsf{minlen}(r_i) \leq \mathsf{minlen}(r_{i+1})$. Alternatives more to the right must be "longer" i.e. have greater or equal minimal length. Thus, if we have the choice between several branches, choosing the rightmost branch means that we can match most of the input with this alternation. We will call regular expressions "branch-sorted", if they have their alternation branches sorted in this way. For longest match, the order of the branches must be reversed, and leftmost branches will be preferred, for reasons explained below.

3 Position Automata

3.1 Basic Definitions

In order to be self-contained, we briefly recall basic definitions of automata (recognizers). A *nondeterministic finite automaton* (nfa) on Σ is a tuple $\mathcal{A} = \langle Q, \Sigma, I, \delta, F\rangle$, where Q is a finite set of states, $I \subseteq Q$ a set of initial states, $\delta : Q \times \Sigma \to 2^Q$ a transition mapping, $F \subseteq Q$ a set of final states.

The language $\mathfrak{L}(\mathcal{A})$ is defined in the usual way by extending δ to $\delta^\star : 2^Q \times \Sigma^\star \to 2^Q$. Deterministic finite automata have a transition mapping $\delta : Q \times \Sigma \to Q$ and only one initial state q_0. Their language is defined by the extended mapping $\delta^\star : Q \times \Sigma^\star \to Q$.

Deterministic automata are obtained from nondeterministic ones by the standard subset construction. Nondeterministic automata are obtained from regular expressions by synthesis (also called automata constructions, or translations). Automata can be turned into regular expressions by analysis.

The traces of an automaton on input $w = a_1 \cdots a_n$ is the set $\mathcal{N}(w)$ of sequences $q_0 \cdots q_n \in Q^\star$ of length $n+1$, with $q_0 \in I$ and $q_i \in \delta^\star(\{q_0\}, a_1 \cdots a_i)$ for all $i \in \{1, \ldots, n\}$.

3.2 Synthesis Algorithm

We have specified pattern matching with a rather semantic, mathematical flavour. Then we motivated the syntactical branch-sortedness of regular expressions with a correspondence between the position in the regular expression and the shortest/longest match. This section ties the knot by providing a link between branch-sorted regular expressions, shortest match, and a particular trace of an automaton.

There is a popular synthesis algorithm that maintains a correspondence between positions of leaves and automaton states, and moreover has the big advantage of avoiding ϵ-transitions altogether. All results in this paper depend on properties of this particular automata construction. It is open whether they can be adapted to further improved constructions like e.g. [14, 12].

We recall the position automata construction, commonly attributed to Berry and Sethi [2](see [16] for a detailed account on a high-performance implementation).

Let us identify a regular expression r with its syntax tree $r : N^\star \to (\Sigma \cup \{\text{SEQ},\text{STAR},\text{OR},\text{EPS}\})$. In the non-trivial case, it has some leaves $w_1, \ldots, w_n$ that are not labelled EPS. In Figure 4, those would be $111, 112, 12, 2, 31$. Every leaf w_i with $t(w_i) = a$ expresses that a must be read, so we can imagine a corresponding automaton state i having all incoming transitions labeled with a. Following this idea, we use the uniquely determined alphabet $\Gamma = \{1, \ldots, n\}$ and mapping $\gamma : \Gamma \to \Sigma, \gamma(i) = r(w_i)$ to obtain a regular expression $r_0 \in \mathsf{RegExp}(\Gamma)$, which we define by its syntax tree:

$$r_0(w) = \begin{cases} i & \text{if } w = w_i \text{ and } \gamma(i) = \sigma \\ r(w) & \text{otherwise} \end{cases}$$

Thus, $(ab|b)ca^\star$ becomes $(12|3)45^\star$. The resulting expression is called linear. We can define the first, last and follow sets of a linear regular expression r_0.

$$\begin{array}{ll} \mathsf{fst}(r_0) & = \{\ i \in \Gamma \mid \exists w \in \Gamma^\star\ .\ iw \in [\![r_0]\!]\ \} \\ \mathsf{lst}(r_0) & = \{\ i \in \Gamma \mid \exists w \in \Gamma^\star\ .\ wi \in [\![r_0]\!]\ \} \\ \mathsf{fol}(r_0, i) & = \{\ j \in \Gamma \mid \exists u, w \in \Gamma^\star\ .\ uijw \in [\![r_0]\!]\ \} \end{array}$$

These sets can be straightfowardly computed simultaneously for all subexpressions of r in time quadratic in n. We are now ready to define $\mathcal{N}_r$, the position automaton of r. It has exactly $n+1$ states, one for each position plus one initial state 0. It can have up to $O(n^2)$ transitions[14].

DEFINITION 1
For $r \in \mathsf{RegExp}(\Sigma)$, the position automaton $\mathcal{N}_r$ is defined as

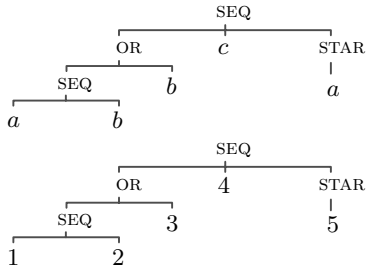


Figure 4: Syntax tree of $(ab|b)ca^\star$ and linearized form

$\langle Q, \Sigma, \{0\}, \delta, \mathsf{lst}(r_0)\rangle$ where $Q = \{0\} \cup \Gamma$ and

$$\begin{array}{ll} \delta(0,a) \ni j & \text{iff } a = \gamma(j) \wedge j \in \mathsf{fst}(r_0) \\ \delta(i,a) \ni j & \text{iff } a = \gamma(j) \wedge j \in \mathsf{fol}(r_0, j) \text{ for all } i \in \Gamma \end{array}$$

Note that by definition the position automaton has the useful property that all transitions that enter a particular state have the same label. We will also need the reversed position automaton $\mathcal{N}^{\mathsf{rev}}$, which is obtained by swapping fst and lst in the above construction and furthermore redefining $\mathsf{fol}(r_0, i) = \{\ j \in \Gamma \mid \exists w, u \in \Gamma^\star\ .\ wjiu \in [\![\mathsf{rex}]\!]\ \}$. It recognizes w^{rev} iff $\mathcal{N}$ recognizes w.

Note that a letter occuring more to the right corresponds to state with greater indices. For branch-sorted regular expressions, this correspondence can be used to get the shortest match.

3.3 An Automaton for Shortest Match

This section states one of the main results, on which disambiguation of regular patterns w.r.t. shortest match is based. We can associate bindings with runs, and the maximal one with the binding for shortest match.

Definition 2
For $w = a_1 \cdots a_n \in [\![r]\!]$, and $\mathcal{N} = \mathcal{N}_r$ its position automaton, a maximal run $\max\mathcal{N}(w)$ is the sequence $q_0 \cdots q_n$ where $q_0 = 0$, $q_n \in F$ and $q_{i-1} = \max \delta^{\mathsf{rev}}(q_i, a_i)$ for all $i \in \{1, \ldots, n\}$ which is maximal w.r.t. $\geq_{\mathsf{right}}$.

Proposition 2
Let w be a word, p be a branch-sorted pattern, and $\mathcal{N} = \mathcal{N}_{\mathbf{vp}(p)}$. For all $s, s' \in \mathsf{Env}(p, w)$, if $\max\mathcal{N}(s) >_{\mathsf{right}} \max\mathcal{N}(s')$ then $s >_{\mathsf{right}} s'$.

This does suggests an naive implementation to find s, which at every ambiguous position searches the maximal state with which it is possible to continue, trying possible choices of x_i and backtracking on failure. In the worst case, this approach traverses the input $O(a * d)$ times, where a is the number of ambiguities and d the maximal number of choices occurring at an ambiguity. In the next section, we show how to find the shortest match of ambiguous patterns in exactly two traversals without backtracking. We use sequential machines to obtain an operational model of pattern matching, and construct the run with the maximal state indices.

4 Sequential Machines

4.1 Basic definitions

Sequential machines represent length-preserving subsequential rational relations. For reasons that will become clear immediately, we reuse $\mathcal{N}$ to denote nsm's in addition to nfa's.

A *nonderministic sequential machine* (nsm) on Σ and Θ is a tuple $\mathcal{N} = \langle Q, \Sigma, \Theta, I, \delta, F\rangle$ similar to an nfa, but with a transition mapping $\delta : Q \times \Sigma \to 2^{\Theta \times Q}$. The transition relation is extended to $\delta^\star$ as before, ignoring the output, and an extended output mapping $\lambda^\star : 2^Q \times \Sigma^\star \to 2^{\Theta^\star}$ can be formulated. The output is discarded if the nsm does not reach a final state.

The traces of an nsm on input $a_1 \cdots a_n$ is a set $\mathcal{N}(w)$ of sequences $q_0 b_1 \cdots q_{n-1} b_n q_n$, with $q_0 \in I$, $q_n \in F$. Symbol b_i being written **after leaving** state q_{i-1} as a result of consuming input symbol a_i. Formally, $\delta(a_i, q_{i-1}) \ni \langle b_i, q_i\rangle$ for each $i \in \{1, \ldots, n\}$, which implies $\delta^\star(\{q_0\}, a_1 \cdots a_i) \ni q_i$ and $\lambda^\star(\{q_0\}, a_1 \cdots a_i) \ni b_1 \cdots b_i$.

An nsm can be seen as an nfa on the alphabet $\Sigma \times \Theta$, if one identifies translation of a word $a_1 \cdots a_k$ into $b_1 \cdots b_k$ and recognition of the word $(a_1 : b_1) \cdots (a_k : b_k)$. This means we can use the synthesis from above to build sequential machines from any regular expression over an alphabet of pairs. To keep notation light, we will also use $a : b$ to denote a pair of elements $a \in \Sigma$ and $b \in \Theta$, and write $\delta(q, a : b) \ni q'$ instead of $\delta(q, a) \ni \langle b, q'\rangle$.

From an nsm one can retrieve all possible outputs(bindings) that a regular expression(pattern) yields given a fixed, accepted input. Instead, a *deterministic sequential machine* (dsm) has a transition mapping $\delta : Q \times \Sigma \to \Theta \times Q$. These correspond to the well known Mealy, Moore machine models. A dsm computes exactly one output for any given, accepted word.

4.2 Translating Patterns

A pattern induces the relation $\mathsf{Env}(p)$ between input words and bindings, but a practical definition of pattern matching should be disambiguated (e.g. respect a match policy). Otherwise put, a pattern should define a *function* from input words to output bindings, not a relation.

The function $\mathbf{h} : V \times \Sigma \to \Sigma \times (V \times \Sigma)$ maps ${}_x a$ to $a : {}_x a$. By extending $\mathbf{h}$ to regular expressions, we turn a pattern $p \in \mathsf{RegPat}(\Sigma, V)$ into a regular expression $\mathbf{vp} \circ \mathbf{h}(p) \in \mathsf{RegExp}(\Sigma \times (V \times \Sigma))$ on an alphabet of pairs $a : {}_x a$. By the considerations in the previous section, applying the position automata construction yields an nsm.

For instance, translating $p = x@a^\star y@aa^\star$ yields the sequential machine $\mathcal{N}_{\mathbf{vp}\circ\mathbf{h}(p)}$ shown in Figure 5. As before, this automaton has a state for every letter occuring in p, plus an initial state. This pattern will be used from now on as a running example.

Proposition 3
Let $p \in \mathsf{RegPat}(\Sigma, V)$ and $\mathcal{N}_{\mathbf{vp}\circ\mathbf{h}(p)}$ its translation. Then $a_1 \cdots a_n \triangleright p \Rightarrow b_1 \cdots b_n$ iff $\mathcal{N}(a_1 \cdots a_n) \ni q_0 b_1 \cdots q_{n-1} b_n q_n$ with $q_n \in F$.

Unlike recognizers, it does not make sense for nondeterministic sequential machines to be "made deterministic" because they can produce several outputs for an input word. As we shall see later, we can recover the set of possible runs (the behavior) of the nsm by applying a subset construction, yielding a sequence of sets of states as output. Then, a second run on this intermediary result is performed in order to choose one among the possible outputs. This scheme follows a long-standing result that states that every sequential rational function can be computed using two sequential machine runs. One can safely postulate that *any* disambiguated form of regular pattern matching can be implemented in this way, because it corresponds to a sequential rational function.

4.3 Ambiguities and decision procedures

After having established a connection between implementations of pattern matching and nsms, we are now in a position to derive definitions of ambiguities and decision procedures.

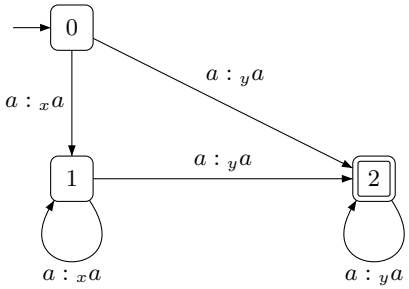


Figure 5: The nsm $\mathcal{N}$ obtained from $x@a^\star y@aa^\star$

Definition 3 (Properties of Patterns)
Let p be a pattern, and $\mathcal{N}_{\mathbf{vp}\circ\mathbf{h}(p)}$ its translation. An *ambiguity* of p is a state q of $\mathcal{N}_p$ which has at least two outgoing transitions $a : {}_{x_i} a, \ldots, a : {}_{x_j} a$ with $x_i \neq x_j$. Thus, p is called

- *ambiguous* if it has at least one ambiguity,
- *unambiguous* if it has none.
- *deterministic* if $\mathcal{N}$ is deterministic, i.e. the range of δ consists only of singleton or empty sets. This implies that p is unambiguous.

For instance, state 0 and 1 the nsm shown in Figure 5 constitute ambiguities. Any well-defined implementation simulates runs of $\mathcal{N}$ at runtime and chooses (e.g. according to a match policy) which transition to take and consequently which x_i to bind to.

Proposition 4 (Decision procedures)
The results lead to decision procedures with time complexity quadratic in $|\Gamma|$, because they can be decided while constructing $\mathcal{N}_p$.

- is p ambiguous?
  For all $i \in \Gamma$, check if $\mathsf{fol}(i, p)$ contains j, l with $\gamma(j) = a : {}_x a$ and $\gamma(l) = a : {}_y a$ with $x \neq y$. Check the same for $\mathsf{fst}(p)$. p is ambiguous iff such a pair j, l is found.
- is p deterministic?
  For all $i \in \Gamma$, check that $|\mathsf{fol}(i, p)| < 1$, and check that $|\mathsf{fst}(p)| < 1$ The time complexity is quadratic in $|\Gamma|$.
- is $[\![p]\!] =$ Check that $\mathcal{N}_p$ has final states.

Checking inclusion requires more effort. Given p, p' with $\mathsf{var}(p) = \mathsf{var}(p')$. If both are deterministic, checking $[\![\mathbf{vp}(p)]\!] \subseteq [\![\mathbf{vp}(p)]\!]$ is done in at most $|\Gamma|^2 * |\Gamma'|^2$, on the product automaton of the two dfa's (we again refer to [16] for a high-performance implementation). In the general case, the worst case complexity is $2^{|\Gamma|^2+|\Gamma'|^2}$, because the nfa's have to be made deterministic before constructing the product automaton.

The position automaton synthesis turns a deterministic pattern directly into a dsm which for every matching w constructs the unique binding s of p in a single traversal. For unambiguous patterns, a subset construction yields a dsm that achieves the same; in these patterns nondeterminism does not lead to ambiguities, hence can be removed.

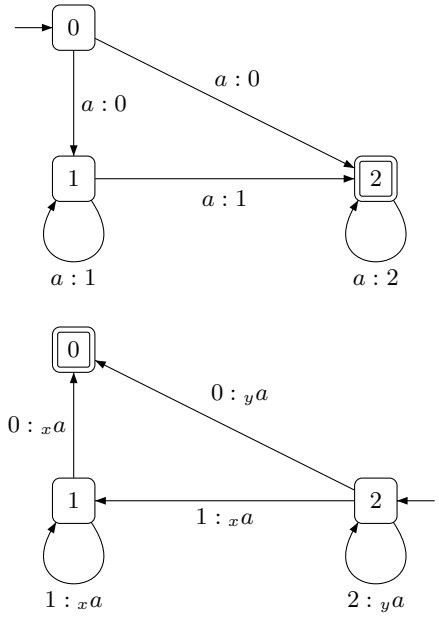


Figure 6: An nsm $\mathcal{L}$ and an nsm $\mathcal{R}$ for $x@a^\star y@aa^\star$

5 Efficient Shortest Match

5.1 Left-Right construction

In the general case, patterns have to be disambiguated using the right-longest match policy. To this end, the nsm described earlier is translated to two deterministic sequential machines.

We separate an nsm $\mathcal{N}$ into two equivalent consecutive nsms $\mathcal{L}$ and $\mathcal{R}$. The first of these reads the input $w = a_1 \cdots a_n$ from left to right and writes the state it is in before making a transition, yielding a word $z = q_0 \ldots q_{n-1}$ and a state q_n. If $q_n \in F$, it is used to choose an initial state of its counterpart $\mathcal{R}$ that reads z from right to left and writes the (reversed) intended output word $s = c_1 \ldots c_n \in (V \times \Sigma)^\star$ (also from right to left). It is easy to see that from an output of $\mathcal{L}$ and the transition table δ of $\mathcal{N}$, we can read off the states occurring in traces $\mathcal{N}(w)$, and thus every trace of $\mathcal{N}$ corresponds to two consecutive traces of $\mathcal{L}$ and $\mathcal{R}$ that produce the same output. The two machines for the running example is shown in Figure 6. Now it is time to remove nondeterminism in $\mathcal{L}$ by a slightly modified subset construction given in Figure 7. The algorithm constructs a sequential machine $\mathsf{det}(\mathcal{L})$. Its states are sets of states of $\mathcal{L}$, which are now printed as output. It considers only reachable states, and ignores the output of $\mathcal{N}$, instead writing the set of states before making a transition. Note that an implementation will assign atomic symbols (e.g. integers) to the reachable subsets of states of $\mathcal{L}$ and use these as output alphabet.

5.2 Determinization of $\mathcal{R}$

States of $\det(L)$ do not correspondend to positions anymore. This correspondence has to be reestablished using $\mathcal{L}$ and $\mathcal{R}$. The task of binding is reduced to taking a trace of $\mathsf{det}(\mathcal{L})$

```
input: nsm L = ⟨Q, q0, δ, F⟩
output:dsm det(L) = ⟨Q̄, q̄0, δ̄, F̄⟩

def next(A, σ) = ⋃_{q∈A} δ(q, σ : __)

initialize Q̄, F̄, δ̄, stack
q̄0 := {q0}
push q̄0 on stack
while( stack not empty )
       pop A from stack
       add A to Q̄
       if( A ∩ F not empty ) then add A to F̄
       for each a ∈ Σ do
              B = next( A, a )
              δ̄(A, a : A) := B
              if( B not in Q ) then push B on stack
```

Figure 7: Constructing $\mathsf{det}(\mathcal{L})$

```
input: dsm det(L) = ⟨Q^det(L), q0^det(L), δ^det(L), F^det(L)⟩
       nsm R = ⟨Q^R, I^R, δ^R, F^R⟩
output:nearly-dsm det(R) = (Q̄, Ī, δ̄, F̄)

initialize Q̄, Ī, stack
for each A ∈ F^det(L)
       choose maximal q from A with q ∈ I^R
       if ⟨q, A⟩ ∉ Q̄ then
            add ⟨q, A⟩ to Q̄
            add ⟨q, A⟩ to Ī
            push ⟨q, A⟩ on stack
while( stack not empty )
       pop ⟨q, A⟩ from stack
       for each B ∈ Q^det(L) with δ^det(L)(B, a : B) = A
              choose maximal q' from B such
                that δ^R(q', q' :_x a) ∋ q for some x
              δ̄(q, B :_x a) := ⟨q', B⟩
              if ⟨q', B⟩ ∉ Q̄ then
                     add ⟨q', B⟩ to Q̄
                     push ⟨q', B⟩ on stack
F̄ := {q0^det(L)}
```

Figure 8: Constructing $\mathsf{det}(\mathcal{R})$

and using it to simulate a run of $\mathcal{N}$ "backwards". When confronted with a choice between several accepting runs of $\mathcal{N}$ we apply the shortest match policy.

The algorithm in Figure 8 which transforms the dsm $\mathcal{R}$ into a dsm $\mathsf{det}(\mathcal{R})$. It reads the trace from $\mathsf{det}(\mathcal{L})$ and writes the output of the one with the maximal state indices of $\mathcal{N}$. The states of $\mathsf{det}(\mathcal{R})$ are pointed sets of states, i.e. pairs $\langle q, A\rangle$ where $q \in A$. We use the state $\langle q, A\rangle$ to simulate that $\mathsf{det}(\mathcal{L})$ was in the state A, which we interpret as being in the state q of $\mathcal{N}$. The initial state is chosen at run-time, depending on the maximal final state of $\mathcal{N}$ that appears in the accepting state of $\mathsf{det}(\mathcal{L})$.

The uniqueness of the maximal run is crucial to obtain the shortest match, as mentioned before. The algorithm constructs a transition mapping of $\mathsf{det}(\mathcal{R})$ which invariably and deterministically builds maximal runs from runs of $\mathsf{det}(\mathcal{L})$:

$$\delta(\langle q, A\rangle, B :_x a) \;=\; \langle q', B\rangle \text{ if}$$

$$\delta^{\mathsf{det}(\mathcal{L})}(B, a : B) = A \text{ and} \qquad (1)$$

$$q' \text{ maximal from } B \text{ such that } \delta^{\mathcal{R}}(q, {}_x a) \ni q' \qquad (2)$$

where $\delta^{\mathsf{det}(\mathcal{L})}$ by is the transition mapping of $\mathsf{det}(\mathcal{L})$ and $\delta^{\mathcal{R}}$ the one of $\mathcal{R}$. For input w, this definition picks the output of $\max\mathcal{L}(w)$ and thus $\max\mathcal{N}(w)$. The first `for` loop of the algorithm computes the several possible initial states as all final states of $\mathsf{det}(\mathcal{L})$, with the greatest possible initial state of $\mathcal{R}$ being the distinguished one. The `while` loop then computes reachable pointed sets of states from the transition mapping of $\mathsf{det}(\mathcal{L})$ and the possible transitions in $\mathcal{R}$. The result of this construction for the running example is displayed in Fig. 9.

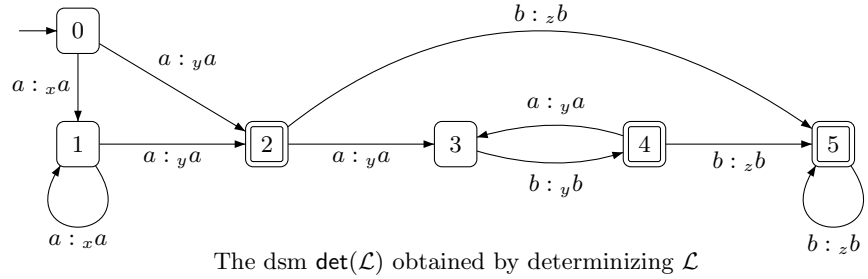
Estimate of the Compile-Time Complexity. Complexity of the quite standard subset construction (used for $\mathcal{L}$) is known [16]. As for nfa, the automaton $\mathcal{L}$ is not necessarily minimal. For the worst-case complexity of determinization of the second transducer. Sets of states and also sets of sets of states are assumed sorted (e.g. using balanced trees). Let $m \in O(2^n)$ be the number of states of $\mathsf{det}(\mathcal{L})$. Since the state set of $\mathcal{R}$ is taken from pointed sets of states, it cannot be more than $m \cdot n$. The body of the while loop takes $O(m * n)$ steps because `choose maximal` tries up to m states q', checking δ^R can fail n times, checking membership in $\overline{Q}$ takes less than $(\log m + n) \in O(n)$. The total number of steps cannot exceed $O(mn(m * (n + n))) = O(m^2n^2)$, or $O(2^{2n})$. The resulting automata are not necessarily minimal, there can be redundant states.

Patterns leading to exponential blowup actually occur in practice, for instance when using adding wildcard-star patterns $\_\_^\star$(see discussion below). But since overall size of patterns is usually small, these algorithms work well in the Scala compiler, which translates the constructed sequential machines to JVM bytecode. The assumption here is that runtime performance is more important than code size and compilation time, which is a common one when compiling pattern matching.

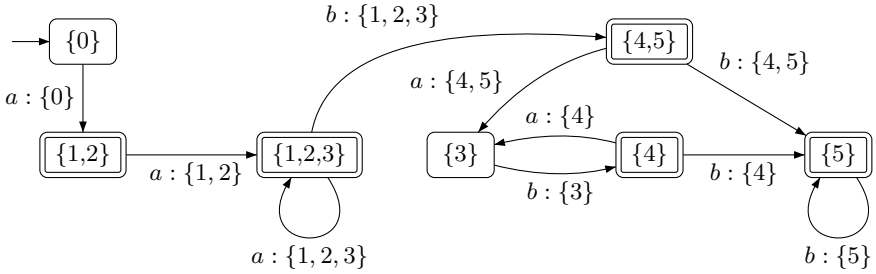
5.3 A larger example

The steps are for compiling regular patterns are sketched in Fig. 10. We conclude the discussion by applying the presented algorithms to a larger example. Consider the pattern $p = x@a^\star y@a(ab)^\star z@b^\star$ given earlier. The position

The nsm $\mathcal{N}$ corresponding to $x@a^\star y@a(ab)^\star z@b^\star$



The dsm $\mathsf{det}(\mathcal{L})$ obtained by determinizing $\mathcal{L}$



The dsm $\mathsf{det}(\mathcal{R})$ obtained by transforming $\mathsf{det}(\mathcal{L})$ and $\mathcal{R}$

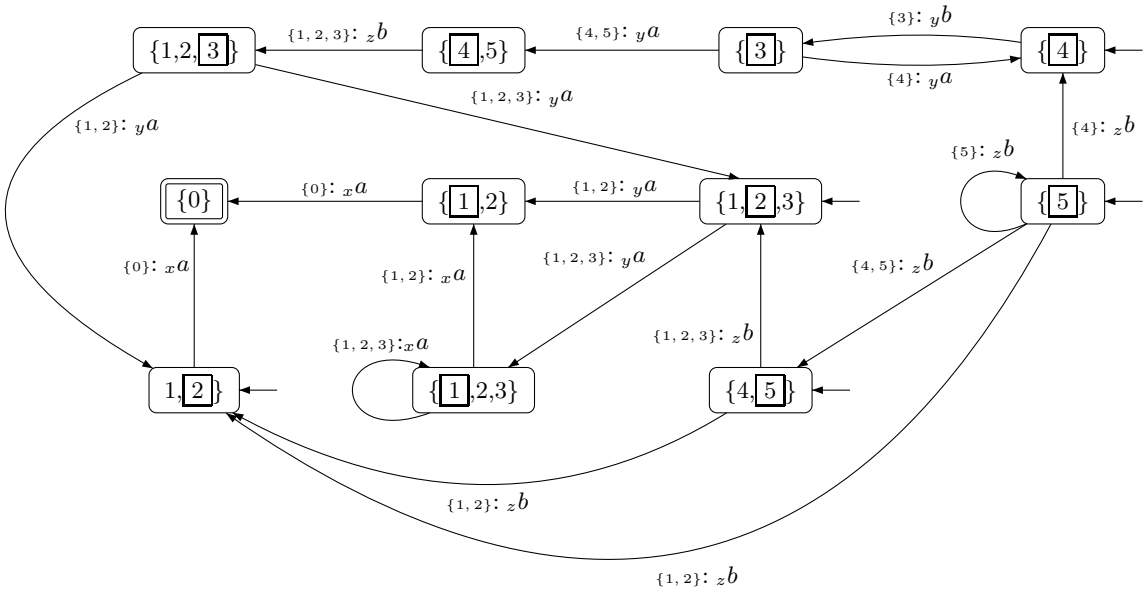


Figure 11: Examples of sequential machines $\mathcal{N}, \mathsf{det}(\mathcal{L}), \mathsf{det}(\mathcal{R})$

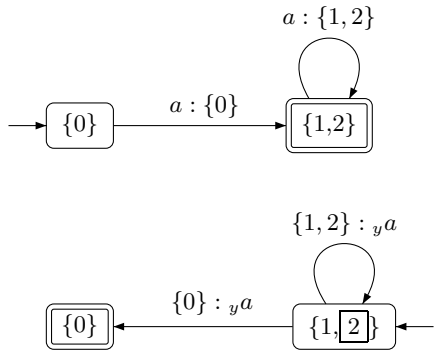


Figure 9: Dsm pair for shortest matches of $x@a^\star y@aa^\star$

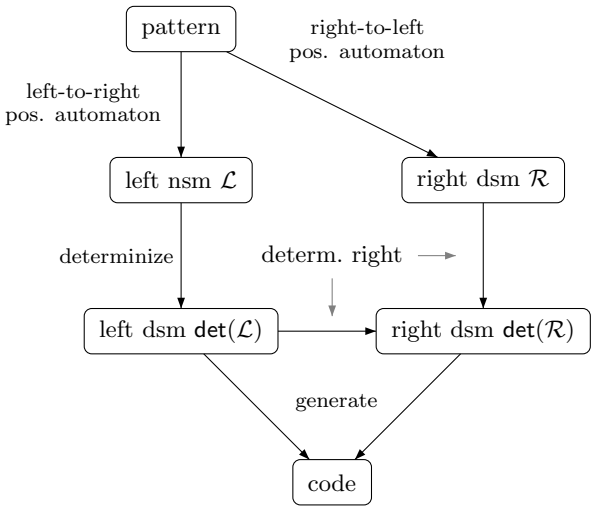


Figure 10: Compiling patterns to sequential machines

automata construction is applied after turning the following expression $\mathbf{vp} \circ \mathbf{h}(p)$, or

$$(a :{}_x a)^\star (a :{}_y a)\big((a :{}_y a)(b :{}_y b)\big)^\star (b :{}_z b)^\star$$

into $1^\star 2(34)^\star 5^\star$. The nsm $\mathcal{N}$, the dsm $\mathsf{det}(\mathcal{L})$ and the dsm $\mathsf{det}(\mathcal{R})$ are shown in Fig. 11. The input word $w = aaabbb$ leads to the following output of $\mathcal{L}$

$$\{0\}\{1,2\}\{1,2,3\}\{1,2,3\}\{4,5\}$$

and ends in the accepting state $\{5\}$. The subsequent run of $\mathsf{det}(\mathcal{R})$ starts by picking the initial state $\langle 5, \{5\}\rangle$, which is abbreviated as $\{\boxed{5}\}$ in the diagram. It takes the output of $\mathsf{det}(\mathcal{L})$ as input and produces from it (from right-to-left) the binding actions ${}_x a_x a_y a_z b_z b_z b$. From this sequence, we obtain the desired right-longest binding:

$$\{x \leftarrow aa, y \leftarrow a, z \leftarrow bbb\}$$

6 Discussion

6.1 Greedy Matching, Greedy Operators

If one does not sort the branches of a pattern according to their minimal length minlen, we get a form of ungreedy matching, because rightmost branches have priority in maximal runs. When looking for minimal runs (which stay "on the left" of the regular expression as long as possible),this yields greedy matching.

Java and Perl offer *ad hoc* operators like "greedy star" and "ungreedy star". Our formal framework can be extended to these. Basically, for states (positions) under a greedy star, we look for minimal states like for longest match, and for states under a nongreedy star, we look for maximal states. Only the construction of $\mathsf{det}(\mathcal{R})$ must be modified, in order to find the unique run $\mathcal{N}(s)$ specified by disambiguated operators.

6.2 Generalizations

To use regular patterns effectively, a couple of generalizations can be made. All fit well with the nondeterministic sequential machine view of pattern matching.

Multiple cases. To use pattern matching in a case distinction, it seems easiest to first recognize which case the input matches, using plain regular expressions, and then use sequential machines for variable binding. It is also possible to construct a product automaton $\mathcal{L}$ of all cases, and adapt $\mathcal{R}$ accordingly. But if no case matches, then the output is discarded, which is inefficient for large inputs.

Operators. Regular expressions can be extended with a wildcard expression $\_\_$ with $[\![\_\_]\!] = \Sigma$, and range patterns $a..b$ with $[\![a..b]\!] = \{c \in \Sigma \mid a \le c \le b\}$ for totally ordered The definitions of fst, lst, fol and the subset constructions are then adapted in a straightforward manner. The dsm $\mathsf{det}(\mathcal{L})$ must then copy the input, in addition to writing its state set.

However, the position automata construction cannot deal with intersection and complement operators [2].

Named expressions can be defined in order to reuse regular expressions like in this example (which looks almost like a lexer specification):

```
let pattern d2    ='0'..'2' in
let pattern d1    ='1'..'9' in
let pattern d0    ='0' | {d1} in
let pattern dayT  ={d1} | ('1'|'2'){d0} | '3'{d2} in
let pattern monT  ={d1} | 1{d2} in
let pattern yearT =2{d0}{d0}{d0} in
  day@{dayT}, '.', mon@{monT}, '.', year@{yearT}
```

Nested binding patterns can be dealt with by letting bindings be sequences over $2^V \times \Sigma$, more precisely over the finite subset $2^{\mathsf{var}(p)} \times \Sigma$. This allows patterns like the following one:

```
date@(day@{dayT}, '.', mon@{monT}, '.', year@{yearT})
```

Variables unter alternation nodes can be allowed, as long as each branch defines the same variables, and the languages of the branches are disjoint and unambiguous, like in

```
  day@{dayT}, '.', mon@{monT}, '.', year@{yearT}
| mon@{monT}, '/', day@{dayT}, '/', year@{yearT}
```

These conditions ensure that such a pattern still defines a function from $\Sigma^\star$ to $(V \times \Sigma)^\star$, when ambiguities in the rest of the pattern are resolved with shortest match.

Tree and Hedge Matching. The theory can be generalized to *hedges* and unranked trees, which are singleton hedges [5, 18]. Labeled hedges $h \in H_\Sigma$ are sequences of labeled trees $a[\ldots]$. Position automata can be defined and turned into deterministic pushdown hedge automata [19]. Patterns are generalized to simple regular hedge expressions:

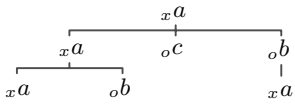
$$\begin{array}{lllll} \mathsf{r} & ::= & \epsilon & [\![\epsilon]\!] & = \{\epsilon\} \\ & & a[r] & [\![a[r]]\!] & = \{a[h] \mid h \in [\![r]\!]\} \quad a \in \Sigma \\ & & \mathsf{r}_1 \cdot \mathsf{r}_2 & [\![\mathsf{r}_1 \cdot \mathsf{r}_2]\!] & = [\![\mathsf{r}_1]\!] \cdot [\![\mathsf{r}_2]\!] \\ & & \mathsf{r}_1 \mid \mathsf{r}_2 & [\![\mathsf{r}_1 \mid \mathsf{r}_2]\!] & = [\![\mathsf{r}_1]\!] \cup [\![\mathsf{r}_2]\!] \\ & & \mathsf{r}^\star & [\![\mathsf{r}^\star]\!] & = [\![\mathsf{r}]\!]^\star \end{array}$$

Note that these do not recognize the full set of regular hedge languages, because depth-iteration is missing [11].

Sequential hedge machines relabel the hedge, transforming H_Σ into $H_{V\times\Sigma}$, the set of hedge bindings. Substitutions are obtained using the mapping

$$\begin{array}{ll} \mathbf{bind}(\epsilon) & = \{x \mapsto \epsilon \mid x \in V\} \\ \mathbf{bind}(h_1 \cdot {}_x a[h_2]) & = \mathbf{bind}(h_1) \\ & \quad \oplus \{x \mapsto f[h_2]\} \oplus \mathbf{bind}(h_2) \end{array}$$

In more conventional terms, patterns for algebraic datatypes are extended to have sequence patterns. These are nodes standing for sequences like arrays or lists that can be conveniently matched with regular patterns. A proof of concept is the current Scala compiler, where these ideas have been implemented [21, 7]. Here is a pattern match that finds the first entry matching `"M"` in a database:



$$\{\ x \ \mapsto\ a[a[a,b],c,b[a]],\ a[a,b],\ a,\ a\ \}$$

Figure 12: XPATH binding for $//a$

```
inp.match {
  case Numbers(_*, Entry("M", num@_), _*) => num
  case _ =>
}
```

All-matches, XPATH, and XEN filters. Leaving aside ambiguities, simple XPATH expressions can be turned into regular hedge patterns with wildcards, one depth-iteration operator for the descendant-axis, and the variable set $\{x, o\}$. Tagging a node with x means the node (and its whole subtree) matches and o means it does not. In this way, a sequential pushdown hedge machine can retrieve all matches from a given input hedge (cf. Figure 12). XEN "filters" are similar to XPATH evaluation without depth-iteration.

6.3 Space and Time Optimization

Results of the subset construction are not always minimal. Standard state reduction can be applied to $\mathcal{R}$ by treating the dsm as a dfa. Reducing $\mathcal{L}$ seems impossible, because then states would no longer be subsets of states of $\mathcal{N}$.

A special case are final sink states generated by a rightmost wildcard pattern $x@\_^\star$. The dsm can stop when entering a sink state, and bind the remaining sequence to x.

Finding shortest matches with two dsms is time-optimal for ambiguous patterns. Shortest match is equivalent to maximal run, and to find a maximal run in the general case, we need to know, for every position in the input, the run of a dfa that goes from left to right (to show which states are reachable), and the run of the reversed dfa from right to left (to see from which states we get to final states). This amounts to two traversals needed for every ambiguous position. Using two dsms merely stores and reuses the result of the first traversal.

Some patterns would be unambiguous if pattern matching would *start* from the right (such as the one in Section 2.3). This can be decided at compile-time to optimize matching for data structures like arrays. However, for iterators and cons-lists matching from right to left requires a traversal and possibly copying, and this leads to complete loss of the gained performance.

7 Conclusion

Based on the position automaton construction, we have defined a language of regular patterns with variable bindings, and gave a compilation scheme which removes ambiguities. The presented results connect long-standing results in formal language theory with recent research in programming languages.

The results are a small step to base new, general forms of pattern matching, that suffer from inherent ambiguity, on sound theoretical foundations. We discussed straightforward generalizations, and sketched the changes necessary for longest (left-longest) match, which is derived from the *minimal* run of $\mathcal{N}$.

We improved a result from Frisch and Cardelli, showing that rewriting regular expressions can be crucial to obtain the longest/shortest match. This suggests systems with regular types be based on semantics, not on syntax of regular expressions. Further research has to be done on the implications of the relation between pattern matching and sequential machines for potential global optimizations in compilers. Consider nested pattern matching statements, where the result ${}_{x_i}a$ of an earlier match can serve as input to another.

Another open issue concerns the suitability of improved nfa constructions [14, 12] for the proposed compilation of pattern matching.

Acknowledgements. I thank Vladimir Gapeyev and Alain Frisch for discussions on their matching implementations. I am also grateful to Sebastian Maneth for discussing tree and hedge matching and helping with the implementation. Finally, I am indebted to Martin Odersky for having provided the opportunity to test these ideas in the SCALA compiler, and to the Hasler foundation for supporting this research with a grant.

References

[1] Véronique Benzaken, Giuseppe Castagna, and Alain Frisch. Cduce: An XML-centric general-purpose language. In *Proceedings of the 8th ACM SIGPLAN International Conference on Functional Programming*, pages 51–63, August 2003.

[2] Gerard Berry and Ravi Sethi. From regular expression to deterministic automata. *Theoretical Computer Science*, 48(1):117–126, 1986.

[3] Jean Berstel. *Transductions and Context-Free Languages*. Teubner Verlag, Stuttgart, 1979.

[4] Niklas Broberg, Andreas Farre, and Josef Svenningsson. Regular expression patterns. In *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming*, 2001.

[5] Anne Brüggemann-Klein, Makoto Murata, and Derick Wood. Regular tree and regular hedge languages over unranked alphabets: Version 1, april 3, 2001. Technical report, Hongkong University of Science and Technology, 2001.

[6] C.C. Elgot and G. Mezei. On relations defined by generalized finite automata. *IBM Journal of Research and Development*, 9:47–65, 1965.

[7] Burak Emir. Extending pattern matching with regular tree expressions for XML processing in Scala. Master's thesis, RWTH Aachen, 2003.

[8] Alain Frisch and Luca Cardelli. Greedy regular expression matching. In *Proceedings of the 31stInternational Colloquium on Automata, Languages and Programming ICALP Turku, Finland*, 2004.

[9] Vladimir Gapeyev, Michael Levin, Benjamin C. Pierce, and Alan Schmitt. XML goes native: Run-time representations for Xtatic. Manuscript.

[10] Vladimir Gapeyev and Benjamin C. Pierce. Regular object types. In *ECOOP'03 - European Conference on Object-Oriented Programming*, volume 2743 of *LNCS*. Springer-Verlag, 2003.

[11] Ferenc Gecseg and Magnus Steinby. *Tree Languages. In: Grzegorz Rozenberg and Arto Salomaa, editors, Handbook of Formal Languages III*, chapter 1, pages 1–68. Springer-Verlag, Heidelberg, 1997.

[12] Christian Hagenah and Anca Muscholl. Computing ϵ-Free NFA from Regular Expressions in $O(n \log^2(n))$ Time. *R.A.I.R.O. Theoretical Informatics and Applications*, 34:257–277, 2000.

[13] Haruo Hosoya and Benjamin Pierce. Regular expression pattern matching for XML. *ACM SIGPLAN Notices*, 36(3):67–80, March 2001.

[14] Juraj Hromkovič, Sebastian Seibert, and Thomas Wilke. Translating regular expression into small ϵ-free nondeterministic finite automata. In *STACS'97 Symposium on Theoretical Aspects of Computer Science*, volume 1200 of *LNCS*, pages 55–66. Springer-Verlag, Heidelberg, 1997.

[15] Michael Y. Levin. Compiling regular patterns. In *Proceedings of the 9th ACM SIGPLAN International Conference on Functional Programming*, 2004. to appear.

[16] O. Matz, A.Miller, A. Potthoff, W. Thomas, and E. Valkema. Report on the program AMoRE. Technical report, Universität Kiel, 1995. also available on http://amore.sourceforge.net.

[17] Eric Meijer, Wolfram Schulte, and Gavin Bierman. Programming with circles, triangles and rectangles. Manuscript, 2003.

[18] Makoto Murata. Hedge automata: a formal model for XML schemata. Manuscript, October 1999.

[19] Andreas Neumann. *Parsing and Querying XML Documents in SML*. PhD thesis, Universität Trier, 1999.

[20] Andreas Neumann and Helmut Seidl. Locating matches of tree patterns in forests. In *18th FST&TCS Foundations of Software Technology and Theoretical Computer Science*, volume 1530 of *LNCS*, pages 134–145. Springer Verlag, Heidelberg, 1998.

[21] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. An overview of the scala programming language. Technical report, EPF Lausanne, 2004.

[22] Naoshi Tabuchi, Eijiro Sumii, and Akinori Yonezawa. Regular expression types for strings in a text processing language (extended abstract). In *Proceedings of TIP'02 Workshop on Types in Programming*, pages 1–18, July 2002.

A Appendix

Proof of Proposition 2: $\max \mathcal{N}(s) >_{\mathsf{right}} \max \mathcal{N}(s')$ yields q_m, q'_m with $q_m > q'_m$ and a common (possibly empty) suffix $z = q_{m+1} \cdots q_n$, i.e.

$$\begin{array}{ll} \max \mathcal{N}(s) & = \cdots q_m z \\ \max \mathcal{N}(s') & = \cdots q'_m z \end{array}$$

This implies a (possibly empty) common suffix

$$\gamma(z) = \gamma(q_{m+1}) \cdots \gamma(q_n)$$

of the bindings s, s'. Let $b = \gamma(q_m) = {}_{x_i}a$ for some ${}_{x_i}a \in V \times \Sigma$. Without loss of generality, we can assume that the bindings have the form

$$\begin{array}{ll} s & = ubb^{len}\gamma(z) \\ s' & = wb'b^{len}\gamma(z) \end{array}$$

for some length *len* (which can also be 0). This means the common suffix $\gamma(z)$ is preceded by at least *len* other letters b in both words, yielding a longer common suffix

If we can now show that $b' = {}_{x_j}a$ with $i > j$ for any length, then $s >_{\mathsf{right}} s'$ follows. That the letter a is the same follows from $\mathsf{proj}(s) = \mathsf{proj}(s')$. We will prove the fact about the variable using the easily shown fact that for any pattern $p = p_1 \cdots p_k$ a maximal trace that binds to x_i must include a unique state q_{last} which is either $q_{\mathsf{last}} = \max \mathsf{lst}(p_j)$ for $j > i$ for nonempty u, w, or $q_{\mathsf{last}} = q_0$ if $u = w = \epsilon$.

So assume the maximal traces of s arrives at q_m after reading the subsequence bb^{len} of s.

$$\begin{array}{ll} \delta^\star(\{0\}, u) & \ni q_{\mathsf{last}} \\ \delta^\star(\{0\}, ubb^{len}) & \ni q_m \\ \delta^\star(\{0\}, wb'b^{len}) & \ni q'_m \end{array}$$

Since r is branch-sorted and $q_m > q'_m$, the number of $b's$ in bb^{len} is greater or equal to the number of b's in $b'b^{len}$. Suppose the number would be the same, and $b' = b$. Then uniqueness of $\max \delta(q_{\mathsf{last}}, b)$, through which both must then pass at the same time and maximality of the runs implies $q_m = q'_m$, a contradiction. So b' must be different, and the order on variables in the pattern yields the desired result $b' = {}_{x_j}a$ for some x_j with $i > j$.□

Lemma 1
Let $s \in [\![p]\!]$ and $\mathcal{N}_{\mathbf{vp}(p)}$ be the position automaton of p. If $s = w_{x_i} a \cdots$ for some $i \in \{1, \ldots, k\}, w \in (X \times \Sigma)^+, X = \{x_1, \ldots, x_{i-1}\} \subseteq V$ then the trace $\max \mathcal{N}_{\mathbf{vp}(p)}(s)$ contains a state $q_{\mathsf{last}} = \max \mathsf{lst}(p_1 \cdots p_j)$ for some $j < i$.

Proof Induction on i.
case 1: Then the assumption on word w cannot be satisfied.
case $i \longrightarrow i + 1$: w ends with ${}_{x_j}b$ for some $j < i$ and some $b \in \Sigma$. Since $s \in [\![p]\!]$, there are states $q \in \mathsf{lst}(p_1 \cdots p_j)$ with $\gamma(q) = {}_{x_j}b$. Every trace in $\mathcal{N}_{\mathbf{vp}(p)}(s)$ goes through such a q. Pick the greatest one of those. If the maximal run would not contain it, it would not be maximal.□

Proof of Proposition 3. For $w = a_1 \cdots a_n \in \Sigma^\star$ and $s = b_1 \cdots b_n \in (V \times \Sigma)^\star$,

$w \triangleright p \Rightarrow s$
iff $\mathsf{proj}(s) = w$ and $s \in [\![p)]\!]$
iff $(a_1 : b_1) \cdots (a_n : b_n) \in [\![\mathbf{vp} \circ \mathsf{h}(p)]\!]$
iff $q_0 b_1 \cdots q_{n-1} b_n q_n \in \mathcal{N}_{\mathbf{vp} \circ \mathsf{h}(p)}(w)$ by correctness of the position automata construction.□