Compiling Regular Patterns to Sequential Machines

Burak Emir
EPFL, 1015 Lausanne, Switzerland
Burak.Emir@epfl.ch

ABSTRACT

Pattern matching combined with regular expressions has many applications including semistructured data matching and lexical analysis in compilers. Variables in patterns allow one to refer to parts of the matching input. But some regular patterns suffer from inherent ambiguity, yielding more than one valid result. A match policy like shortest or longest match can disambiguate such patterns.

In this paper, we show that regular pattern matching corresponds to sequential transduction. We derive straightforward ways to optimally compile regular patterns to sequential machines and to decide when regular patterns are unambiguous. Unambiguous patterns can be matched in a single traversal of the input. Ambiguities in patterns correspond to nondeterminism in sequential machines. Applying the match policy optimally yields two deterministic sequential machines, which produce the shortest match in two consecutive runs.

Categories and Subject Descriptors

D.3.1 [**Formal Definitions and Theory**]: Semantics; D.3.3 [**Language Constructs and Features**]: Patterns

Keywords

Regular patterns, matching, sequential machines

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 semistructured data in languages like XDUCE [12] and CDUCE [1]. For general-purpose programming, regular pattern matching is used in XTATIC [10, 9], HARP [4] (an extension of HASKELL) and SCALA [16]. XEN filters [15] and XPATH expressions are related constructs. Regular pattern matching in various forms is also being applied to DNA sequences or TCP network packets, and for lexical analysis in parsers. Language support for regular patterns can be highly desirable in such domains.

Regular patterns are a natural generalization of pattern matching as known from ML and HASKELL. Here is an example on how to succinctly query an email for its sender:

```
email.match {
  case ('F','r','o','m',':',x@__*,'\n',__*) => x
}
```

We use the convention that `__` is a wildcard pattern and that a binding pattern $v@p$ matches everything p matches, binding the result to the variable v. The pattern should bind everything between "`From:`" and the first newline character to the variable x. But the iterated wildcard pattern `__*` matches arbitrarily many arbitrary characters, so the binding pattern $x@$`__*` might possibly stretch far beyond the first newline. The pattern is ambiguous; several values for x are possible.

Ambiguity can be removed by imposing a match policy. We are interested in the *shortest* match here because we want binding to stop at the *first* newline character. Ambiguities appear also in absence of wildcards, as can be seen from the minimal example $(x@\texttt{`a`}^\star,\ y@\texttt{`a`}^\star)$.

In this paper, we give a formal model of shortest match and derive decision procedures and algorithms from it. The essential idea is that binding to a variable can be seen as tagging parts of the input with the variable. Let us agree on writing ${}_x a$ for appending input element a to variable x. Then matching "`From:jj@foo.net` $\begin{smallmatrix}\mathtt{N}\\ \mathtt{L}\end{smallmatrix}$ `To:...`" against the email pattern above is conceptually the same as transforming the input string to

$$\texttt{"From:}\,{}_x\texttt{j}\,{}_x\texttt{j}\,{}_x@\,{}_x\texttt{f}\,{}_x\texttt{o}\,{}_x\texttt{o}\,{}_{x\cdot x}\texttt{n}\,{}_x\texttt{e}\,{}_x\texttt{t}\ \begin{smallmatrix}\mathtt{N}\\ \mathtt{L}\end{smallmatrix}\ \texttt{To:...".}$$

The desired substitution $\{x \mapsto$ "`jj@foo.net`"$\}$ can be read off the output (do not confuse the character @ with the binding operator @). The shortest match and the semantics of pattern matching can be concisely specified by reasoning on such annotated strings. We obtain a natural formulation of pattern matching as length-preserving rational transduction.

Contributions. The contributions of this paper are (1) the (apparently first) sequential machine formulation of regular pattern matching, (2) algorithms based on this model that generate code with linear runtime complexity, (3) a decision procedure for unambiguous patterns, (4) an intuitive account on match policies and position automata. The accompanying report [7] discusses various generalizations and applications and should be consulted for details.

The technique discussed in this paper is implemented in the reference SCALA compiler [6] and used in practice. The results should help implement regular pattern matching, relate different approaches and proof techniques, and prevent implementors from reinventing the wheel.

Related work. Ambiguities are first mentioned by Hosoya, Vouillon and Pierce [12]. Disambiguation of patterns is specified rigorously by Tabuchi *et al* [17].

SAC'05 March 13-17, 2005, Santa Fe, New Mexico, USA

Frisch and Cardelli [8] implement disambiguated pattern matching. Their goals and assumptions differ though: (1) They consider "greedy matching", a local approximation of the longest match. We show how the longest match can be obtained without approximation with the same runtime complexity. (2) Their approach is presented in a framework that prohibits the rewriting of regular expressions. In our approach however, we show that rewriting turns out to be *indispensable* for longest/shortest match. Moreover, the correspondence with sequential machines presented here seems rather more intuitive than the sets of structured values they employ.

Broberg, Farre and Svenningsson [4] add regular patterns to HASKELL. They handle ambiguities with greedy and non-greedy operators, and by returning a list of all matches in other cases. This poses challenges to efficient implementation, however.

Disambiguated operators are also present in PERL and JAVA. We sketch how our approach can be extended to disambiguated operators without sacrificing efficiency.

Organization of the paper The accompanying technical report [7] contains the complete definitions, proofs and a larger example. We introduce patterns and bindings in Section 2. We then recall position automata and put them in correspondence to shortest match in Section 3. In Section 4, we introduce sequential machines. In Section 5, we turn to efficient compilation to these sequential machines, followed by conclusion, acknowledgment and references.

2. REGULAR PATTERNS

2.1 Patterns and Bindings

Patterns and their denotation are defined as follows. The definitions use a mapping $\mathbf{vp}$ which is explained below.

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

We write $\mathsf{RegExp}(\Sigma)$ for the set of regular expressions over Σ, and $\mathsf{RegPat}(\Sigma, V)$ for the set for regular patterns over alphabet Σ and variables V. 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 ordered set $\{x_1, \ldots, x_k\}$ is denoted $\mathsf{var}(p)$. If not otherwise mentioned, we always talk about a fixed pattern $p = x_1@r_1 \cdots x_k@r_k$.

The function $\mathbf{vp}$ pushes variables down to the leaves. A pattern over Σ and V becomes an expression over $\mathsf{var}(p) \times \Sigma$. For instance, writing ${}_x a$ instead of $\langle x, a \rangle$,

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

Patterns are thus regular expressions over $\mathsf{var}(p) \times \Sigma$. Sequences $s \in (V \times \Sigma)^\star$ can be seen as substitutions built incrementally. We call such sequences *bindings*, and an element ${}_x a$ a *binding action*. The projection $\mathsf{proj} : V \times \Sigma \to \Sigma$ can be extended to sequences and regular expressions. A mapping $\mathbf{bind}$ maps sequences of binding actions to substitutions. 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 $\mathsf{proj}(s_i) = \mathbf{bind}(s)(x_i)$. The subsequence s_i is called "binding for variable x_i".

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. A word w matches $x_1@r_1 \cdots x_k@r_k$ if $w \in [\![r_1 \cdots r_k]\!]$ and we can find a binding s with $\mathbf{bind}(s)(x_i) \in [\![r_i]\!]$ for each i. Since bindings are annotated input words, we can view the problem as transforming input $w \in \Sigma^\star$ into output $s \in (V \times \Sigma)^\star$.

To specify the longest and shortest match, it is enough to focus on the outputs, using transitions of the form ${}_x a$. These automata are considered to generate bindings; in practice, they will modify the environment at runtime (e.g. by modifying a pointer-structure on the heap).

A word w, a pattern p and a binding s are in the ternary matching relation $w \triangleright p \Rightarrow s$ (pronounced "w matches p yielding s") if $\mathsf{proj}(s) = w$ and $s \in [\![p]\!]$.

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 write $\mathsf{Env}(p, w) = \{\ s \mid w \triangleright p \Rightarrow s\ \}$ for the set of possible bindings for pattern p and word w. This set can have more than one element. For instance, for the word a^m, the pattern $x@a^\star y@a^\star$ yields $m + 1$ possible bindings.

Patterns which for a matching word yield more than one binding are ambiguous. In such a case, a match policy picks the one that should be used.

2.3 Shortest Match

The shortest (or right-longest) match means starting from rightmost binding pattern and assigns the longest *possible* subsequence to its variable. The longest (or left-longest) match policy is defined symmetrically from the left. In the example $x@a^\star y@a^\star$, shortest match will assign the whole input to y. Consider for instance matching $aaabbb$ against $x@a^\star y@a(ab)^\star z@b^\star$:

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

Intuitively, matching requires backtracking. In this particular example left-longest and right-longest match coincide, but obtaining the left-longest match naively requires backtracking. For symmetry reasons, we focus on shortest (or right-longest) match from now on.

A total order $s >_{\mathsf{right}} s'$ on $\mathsf{Env}(p, w)$ tells us whether a binding is right-longer. We can formally define it as the reversed lexicographical order on the lengths of the s_i, with the rightmost position being most significant. 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. We write $>_{\mathsf{right}}$ also for the reversed lexicographical order on $\mathbb{N}^n$.

In addition to these definitions, we need a technical property to sort the branches of an alternation. The *minimal length* $\mathsf{minlen}(r)$ describes the minimal length of a sequence that matches r. For right-longest match, we can rewrite a regular expression such that for every alternation $r_1| \ldots |r_m$ we have $\mathsf{minlen}(r_i) \leq \mathsf{minlen}(r_{i+1})$. We will call regular expressions "branch-sorted", if they have their alternation branches sorted in this way. Branch-sortedness is necessary, but not yet sufficient to obtain shortest match.

3. POSITION AUTOMATA

3.1 Synthesis Algorithm

We briefly recall the position automata construction (commonly attributed to Berry and Sethi [2]) which does without ϵ-transitions and maintains a correspondence between positions of leaves in the syntax tree of a regular expression and automaton states. All results depend on the properties of the automata resulting from this particular construction. It is open whether they can be adapted to further improved constructions like e.g. [13, 11]. For details on implementing it, we refer the reader to the report [7] or the manual of the AMoRE automata library [14] .

The basic idea is to traverse the syntax tree of the regular expression in preorder, assigning numbers to the leaves (see Figure 1). A mapping $\gamma : \Gamma \to \Sigma$ maps numbers back to letters. Then first, last and follow sets are computed simultaneously for all subexpressions of r in time quadratic in n. The position automaton $\mathcal{N}_r$ of a regular expression r has exactly $n + 1$ states, one for each position plus one initial state 0. It can have up to $O(n^2)$ transitions [13].

DEFINITION 1
For $r \in \mathsf{RegExp}(\Sigma)$, the position automaton $\mathcal{N}_r$ is defined as $\langle Q, \Sigma, \{0\}, \delta, \mathsf{lst}(r_0)\rangle$ where $Q = \{0\} \cup \Gamma$ and

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

By definition, all transitions that enter a particular state have the same label. The reversed position automaton $\mathcal{N}^{\mathsf{rev}}$ can be obtained by swapping fst and lst in the above construction and furthermore redefining follow. It recognizes w^{rev} iff $\mathcal{N}$ recognizes w. Note that a letter occurring more to the right corresponds to a state with a greater index. For branch-sorted regular expressions, this correspondence can be used to get the shortest match.

3.2 Maximal Run for Shortest Match

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

DEFINITION 2
For $w \in [\![r]\!]$, and $\mathcal{N}_r$ the position automaton of r, a maximal run $\max\mathcal{N}(w)$ is the sequence $q_0 \cdots q_n$ which is maximal with respect to $\geq_{\mathsf{right}}$.

PROPOSITION 2
Let w be a word, p be a branch-sorted pattern, and $\mathcal{N} = \mathcal{N}_{\mathbf{vp}(p)}$. For all $s, s' \in \mathsf{Env}(p, w)$, it holds that

$$\max\mathcal{N}(s) >_{\mathsf{right}} \max\mathcal{N}(s') \text{ implies } s >_{\mathsf{right}} s'.$$

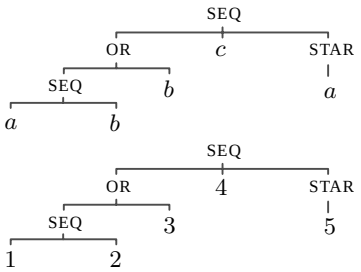

Figure 1: Syntax tree of $(ab|b)ca^\star$ and linearized form

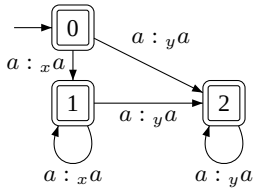

Figure 2: The nsm $\mathcal{N}$ obtained from $x@a^\star y@a^\star$

A naive strategy to find the right-longest s is to find the maximal states that lead to a final state using backtracking. In the next section, we show how to do without backtracking using two traversals of the input.

4. SEQUENTIAL MACHINES

4.1 Basic Definitions

Sequential machines represent length-preserving subsequential rational relations [3]. A *nondeterministic 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.

We have reused $\mathcal{N}$ to denote nsms in addition to nfas. An nsm can in fact 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 construction 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

To obtain nsms from patterns, we extend a mapping $\mathbf{h}$ that duplicates input letters, i.e. sends ${}_x a$ to $a : {}_x a$. If $\mathbf{h}$ is applied after $\mathbf{vp}$, a pattern $p \in \mathsf{RegPat}(\Sigma, V)$ is turned into a regular expression $\mathbf{vp} \circ \mathbf{h}(p) \in \mathsf{RegExp}(\Sigma \times (V \times \Sigma))$ on an alphabet of pairs $a : {}_x a$. As explained before, applying the position automata construction yields an nsm.

For instance, translating $p = x@a^\star y@a^\star$ yields the sequential machine $\mathcal{N}_{\mathbf{vp}\circ\mathbf{h}(p)}$ shown in Figure 2. This pattern will be used from now on as a running example.

PROPOSITION 3
Let $p \in \mathsf{RegPat}(\Sigma, V)$ and $\mathcal{N} = \mathcal{N}_{\mathbf{vp}\circ\mathbf{h}(p)}$ its translation. Then $w \triangleright p \Rightarrow s$ iff $\mathcal{N}$ can accept w producing output s.

Unlike recognizers, nsms cannot be "made deterministic" because they can produce several outputs for an input word. But we can recover the set of possible runs as an intermediary result and then choose one among the possible outputs with a second run. This scheme follows a long-standing result by Elgot and Mezei [5]. *Any* disambiguated form of regular pattern matching (i.e. sequential rational function) can be implemented in this way, in particular also PERL greedy and ungreedy operators.

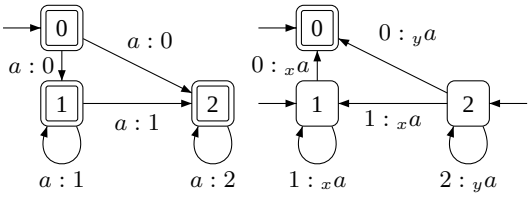

Figure 3: Nsm $\mathcal{L}$ and nsm $\mathcal{R}$ for $x@a^\star y@a^\star$

4.3 Ambiguities and Decision Procedures

Having established a connection between patterns and nsms, we can now derive definitions of ambiguities and decision procedures.

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 $\mathcal{N}_{\mathbf{vp}\circ\mathbf{h}(p)}$ with at least two outgoing transitions $a : {}_{x_i}a, a : {}_{x_j}a$ where $x_i \neq x_j$. 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 2 constitute ambiguities. Any correct implementation of regular pattern matching has to simulate runs of $\mathcal{N}_{\mathbf{vp}\circ\mathbf{h}(p)}$ at runtime and choose (e.g. according to a match policy) which transition to take and consequently which x_i to bind to.

PROPOSITION 4 (DECISION PROCEDURES)
We derive decision procedures with time complexity quadratic in $|\Gamma|$ (they are effected during automata construction).

- is p ambiguous? For all $i \in \Gamma$, check if $\mathsf{fol}(i, p)$ contains j, l with $\gamma(j) = a : {}_{x}a$ and $\gamma(l) = a : {}_{y}a$ where $x \neq y$. 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 $|\mathsf{fst}(p)| < 1$.

We can also check inclusion for p_1, p_2 with $\mathsf{var}(p_1) = \mathsf{var}(p_2)$ using the standard product construction on the $\mathcal{N}_{\mathbf{vp}(p_i)}$, with worst case complexity $|\Gamma|^2 * |\Gamma'|^2$ for deterministic patterns, and $2^{|\Gamma|^2+|\Gamma'|^2}$ in the general case (the nfas have to be made deterministic before). The position automaton construction 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.

5. EFFICIENT SHORTEST MATCH

For the remaining patterns, we have to apply the shortest match policy. To this end, the nsm $\mathcal{N}$ obtained from a pattern is split into two nsms $\mathcal{L}$ and $\mathcal{R}$ (see Figure 3). $\mathcal{L}$ reads the input from left to right and writes the state it is in before making a transition, yielding a word $q_0 \ldots q_{n-1}$ and the final state q_n. Based on $q_n \in F$, an initial state of its counterpart $\mathcal{R}$ is chosen that reads z from right to left and writes the intended output s from right to left. Thus, only $\mathcal{L}$ is nondeterministic. It is easy to see that these two nsms accept the same inputs yielding the same outputs as the original nsm.

input: $\mathcal{L} = \langle Q, q_0, \delta, F\rangle$ output:$\mathsf{det}(\mathcal{L}) = \langle \overline{Q}, \overline{q_0}, \overline{\delta}, \overline{F}\rangle$

initialize $\overline{Q}, \overline{F}, \overline{\delta}$, stack
$\overline{q_0} := \{q_0\}$
push $\overline{q_0}$ on stack
while(stack not empty)
  pop A from stack
  add A to $\overline{Q}$
  if($A \cap F$ not empty) then add A to $\overline{F}$
  for each $a \in \Sigma$ do
    $B = \bigcup_{q\in A} \delta(q, a : \_)$ // *ignore output*
    $\overline{\delta}(A, a : A) := B$
    if(B not in Q) then push B on stack

Figure 4: Constructing $\mathsf{det}(\mathcal{L})$

input: dsm $\mathsf{det}(\mathcal{L}) = \langle Q^{\mathsf{det}(\mathcal{L})}, q_0^{\mathsf{det}(\mathcal{L})}, \delta^{\mathsf{det}(\mathcal{L})}, F^{\mathsf{det}(\mathcal{L})}\rangle$
  nsm $\mathcal{R} = \langle Q^{\mathcal{R}}, I^{\mathcal{R}}, \delta^{\mathcal{R}}, F^{\mathcal{R}}\rangle$
output:dsm $\mathsf{det}(\mathcal{R}) = \langle \overline{Q}, \overline{I}, \overline{\delta}, \overline{F}\rangle$

initialize $\overline{Q}, \overline{I}$, stack
for each $A \in F^{\mathsf{det}(\mathcal{L})}$
  choose maximal q from A with $q \in I^{\mathcal{R}}$
  if $\langle q, A\rangle \notin \overline{Q}$ then
    add $\langle q, A\rangle$ to $\overline{Q}$
    add $\langle q, A\rangle$ to $\overline{I}$
    push $\langle q, A\rangle$ on stack
while(stack not empty)
  pop $\langle q, A\rangle$ from stack
  for each $B \in Q^{\mathsf{det}(\mathcal{L})}$ with $\delta^{\mathsf{det}(\mathcal{L})}(B, a : B) = A$
    choose maximal q' from B such
      that $\delta^{\mathcal{R}}(q', q' : {}_{x}a) \ni q$ for some x
    $\overline{\delta}(q, B : {}_{x}a) := \langle q', B\rangle$
    if $\langle q', B\rangle \notin \overline{Q}$ then
      add $\langle q', B\rangle$ to $\overline{Q}$
      push $\langle q', B\rangle$ on stack
$\overline{F} := \{q_0^{\mathsf{det}(\mathcal{L})}\}$

Figure 5: Constructing $\mathsf{det}(\mathcal{R})$

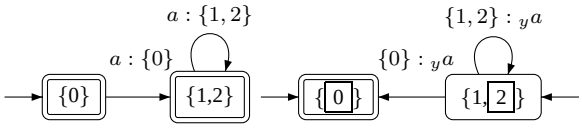

Figure 6: $\mathsf{det}(\mathcal{L})$ and $\mathsf{det}(\mathcal{R})$ for $x@a^\star y@a^\star$

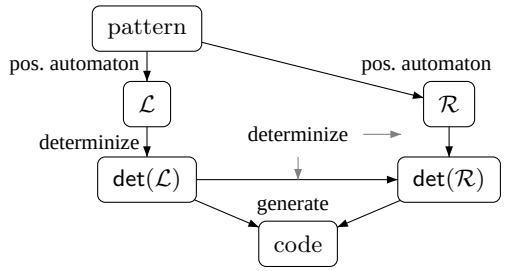

Figure 7: Compiling patterns to sequential machines

Nondeterminism in $\mathcal{L}$ is removed by a slightly modified subset construction given in Figure 4. It constructs a sequential machine $\mathsf{det}(\mathcal{L})$ that prints sets of states as outputs.

Then the algorithm in Figure 5 constructs a dsm $\mathsf{det}(\mathcal{R})$ from $\mathcal{R}$ and $\mathsf{det}(\mathcal{L})$. Its states are *pointed* sets of states $\langle q, A\rangle$ which reconstruct the maximal run and output of $\mathsf{det}(\mathcal{L})$. By the propositions from above, this yields the shortest match. Figure 6 contains the output for the running example $x@a^\star y@a^\star$. More examples can be found in the technical report, and in the SCALA documentation. The latter shows how to apply this versatile construct in practice.

The subset construction for $\mathsf{det}(\mathcal{L})$ might lead to state space explosion, but we assume that runtime performance is more important than code size and compilation time. Since the size of patterns is usually small, these algorithms work well in the SCALA compiler. Figure 7 shows the full compilation scheme.

6. CONCLUSION

We showed how to compile regular patterns with a match policy to sequential machines. Rewriting regular expressions is crucial in our approach. The longest/shortest match of ambiguous patterns is obtained in two runs, which is optimal in the general case.

The accompanying report discusses straightforward generalizations, and sketches changes necessary for longest (left-longest) match, which is derived from the *minimal* run.

Further research has to be done on whether sequential machines can be used in compiler optimizations.

Acknowledgements. Thanks to all reviewers for helpful comments. I thank Vladimir Gapeyev, Alain Frisch and Sebastian Maneth for discussions on regular pattern matching. 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.

7. REFERENCES

[1] V. Benzaken, G. Castagna, and A. Frisch. Cduce: An XML-centric general-purpose language. In *Proc. 8th ICFP*, pages 51–63, Aug. 2003.

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

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

[4] N. Broberg, A. Farre, and J. Svenningsson. Regular expression patterns. In *Proc. 10th ICFP*, 2001.

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

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

[7] B. Emir. Compiling regular patterns to sequential machines. Technical Report IC/2004/72, EPF Lausanne, 2004.

[8] A. Frisch and L. Cardelli. Greedy regular expression matching. In *Proc. 31st ICALP*, 2004.

[9] V. Gapeyev, M. Levin, B. C. Pierce, and A. Schmitt. XML goes native: Run-time representations for Xtatic. Manuscript.

[10] V. Gapeyev and B. C. Pierce. Regular object types. In *Proc. ECOOP*, volume 2743 of *LNCS*. Springer, 2003.

[11] C. Hagenah and A. 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.

[12] H. Hosoya and B. Pierce. Regular expression pattern matching for XML. *ACM SIGPLAN Notices*, 36(3):67–80, Mar. 2001.

[13] J. Hromkovič, S. Seibert, and T. Wilke. Translating regular expression into small ϵ-free nondeterministic finite automata. In *STACS*, volume 1200 of *LNCS*. Springer Verlag, 1997.

[14] O. Matz, A.Miller, A. Potthoff, W. Thomas, and E. Valkema. Report on the program AMoRE. Technical report, Universität Kiel, 1995.

[15] E. Meijer, W. Schulte, and G. Bierman. Programming with circles, triangles and rectangles. Manuscript, 2003.

[16] M. Odersky et al. An overview of the Scala programming language. Technical Report IC/2004/64, EPF Lausanne, 2004.

[17] N. Tabuchi, E. Sumii, and A. Yonezawa. Regular expression types for strings in a text processing language (extended abstract). In *Proc. TIP'02 Workshop on Types in Programming*, pages 1–18, July 2002.

About the Author

Burak Emir is a student author who graduated in 2003 from the Rheinisch-Westfälische Technische Hochschule (RWTH) Aachen, Germany. He wrote his master thesis under supervision of Martin Odersky at the Ecole Polytéchnique Fédérale in Lausanne (EPFL), Switzerland. He joined his group immediately afterwards and started pursueing a PhD. His reseach interest lies in programming language abstractions for semistructured data. He maintains and refines the pattern matching facilities and the XML library functions of the Scala reference implementation, and recently started developing a programming language for statically typed XML processing based on regular pattern matching.