Greedy Regular Expression Matching

Alain Frisch[1,2,⋆] and Luca Cardelli[3]

[1] École Normale Supérieure (Paris)
[2] École Nationale Supérieure des Télécommunications (Paris)
[3] Microsoft Research

Abstract. This paper studies the problem of matching sequences against regular expressions in order to produce structured values.

1 Introduction

Regular expressions play a key role in XML [W3C00]. They are used in XML schema languages (DTD, XML-Schema [W3C01], Relax-NG, ...) to constrain the possible sequences of children of an element. They naturally lead to the introduction of *regular expression types* and *regular expression patterns* in XML-oriented functional languages (XDuce [HVP00,HP03,Hos01], XQuery [BCF+03b], ℂDuce [BCF03a]). These works introduce new kinds of questions and give results in the theory of regular expression and regular (tree) languages, such as efficient implementation of inclusion checking and boolean operations, type inference for pattern matching, checking of ambiguity in patterns [Hos03], compilation and optimization of pattern matching [Lev03,Fri04], etc...

Our work is a preliminary step in introducing similar ideas to imperative or object-oriented languages. While Xtatic [GP03] uses a uniform representation of sequences, we want to represent them with structured data constructions that provide more efficient representation and access. As in XDuce, our types are regular expressions: we use $\times$, $+$, *, ε to denote concatenation, alternation, Kleene star and the singleton set containing the empty sequence. But our types describe not only a set of possible sequences, but also a concrete structured representation of values. As in the Xen language [MS03], we map structural types to native .NET CLR [ECM02] types, however we define subtyping on the basis of *flattened* structures, in order to support natural semantic properties of regular language inclusion. For instance, $(\texttt{int} \times \texttt{int})$ is a set-theoretic subtype of $\texttt{int}^*$, but we need a coercion to use a value of the former where a value of the latter is expected, because the runtime representations of the two types are different. Such a coercion can always be decomposed (at least conceptually) in two phases: flatten the value of the subtype to a uniform representation, and then match that flat sequence against the super type. The matching process is a generalization of pattern matching in the sense of XDuce [HP01].

This paper does not propose a language design. Instead, we study the theoretical problem of matching a flat sequence against a type (regular expression);

⋆ This work was supported by an internship at Microsoft Research.

the result of the process is a structured value of the given type. In doing so, one must pay attention to ambiguity in matching. Our contributions, thus, are in noticing that: (1) A disambiguated result of parsing can be presented as a data structure that does not contain ambiguities. (2) There are problematic cases in parsing values of star types that need to be disambiguated (Prop. 1). (3) The disambiguation strategy used in XDuce and ℂDuce pattern matching can be characterized mathematically by what we call greedy regular expression matching. (4) There is a linear time algorithm for the greedy matching.

There is a rich literature on efficient implementation of regular expression pattern matching [Lau01,Kea91,DF00]. There is a folklore problem with expression-based implementations of regular expression matching: they don't handle correctly the case of a regular expression t^* when t accepts the empty word. Indeed, an algorithm that would naively follow the expansion $t^* \rightsquigarrow (t \times t^*) + \varepsilon$ could enter an infinite loop. Harper [Har99] and Kearns [Kea91] propose to keep the naive algorithm, but to use a first pass to rewrite the regular expressions so as the remove the problematic cases. For instance, let us consider the regular expression $t = (a^* \times b^*)^*$. We could rewrite it as $t' = ((a \times a^*) \times b^* + (b \times b^*))^*$. In general, the size of the rewritten expression may be exponential in the size of the original expression. Moreover, changing the regular expression changes the type of the resulting values, and the interaction with the disambiguation policy (see below) is not trivial. Therefore, we do not want to rewrite the regular expressions. Another approach is to patch the naive recognition algorithm to detect precisely the problematic case and cut the infinite loop [Xi01]. This is an *ad hoc* way to define the greedy semantics in presence of problematic regular expressions.

Our approach is different since we want to axiomatize abstractly the disambiguation policy, without providing an explicit matching algorithm. We identify three notions of problematic words, regular expressions, and values (which represent the ways to match words), relate these three notions, and propose matching algorithms to deal with the problematic case.

2 Notations

Sequences. For any set X, we write X^* for the set of finite sequences over X. Such a sequence is written $[x_1; \dots; x_n]$. The empty sequence is $[]$. We write $x :: s$ for the sequence obtained by prepending x in front of s and $s :: x$ for the sequence obtained by appending x after s. If s_1 and s_2 are sequences over X, we define $s_1 @ s_2$ as their concatenation. We extend these notations to subsets of X^* with $x :: X_1 = \{x :: s \mid s \in X_1\}$ and $X_1 @ X_2 = \{s_1 @ s_2 \mid s_i \in X_i\}$.

Symbols, words. We assume to be given a fixed alphabet Σ, whose elements are called symbols (they will be denoted with c,c_1,...). Elements of Σ^* are called words. They will be denoted with w, w_1,w',...

Types. The set of types is defined by the following inductive grammar:

$$t \in T \quad ::= \quad c \mid (t_1 \times t_2) \mid (t_1 + t_2) \mid t^* \mid \varepsilon$$

Values. The set of values $\mathcal{V}(t)$ of type t is defined by:

$$\begin{array}{ll}
\mathcal{V}(c) & := \{c\} \\
\mathcal{V}(t_1 \times t_2) & := \{(v_1, v_2) \mid v_i \in \mathcal{V}(t_i)\} \\
\mathcal{V}(t_1 + t_2) & := \{e : v \mid e \in \{1,2\}, v \in \mathcal{V}(t_e)\} \\
\mathcal{V}(t^*) & := \{[v_1; \ldots; v_n] \mid v_i \in \mathcal{V}(t)\} \\
\mathcal{V}(\varepsilon) & := \{\varepsilon\}
\end{array}$$

The symbol ε as a value denotes the sole value of ε as a type. We will use the letter σ to denote elements of $\mathcal{V}(t^*)$. Note that the values are structured elements, and no flattening happen automatically.

The flattening $\texttt{flat}(v)$ of a value v is a word defined by:

$$\begin{array}{ll}
\texttt{flat}(c) & := [c] \\
\texttt{flat}((v_1, v_2)) & := \texttt{flat}(v_1)@\texttt{flat}(v_2) \\
\texttt{flat}(e : v) & := \texttt{flat}(v) \\
\texttt{flat}([v_1; \ldots; v_n]) & := \texttt{flat}(v_1)@\ldots@\texttt{flat}(v_n) \\
\texttt{flat}(\varepsilon) & := []
\end{array}$$

We write $\texttt{flat}(t) = \{\texttt{flat}(v) \mid v \in \mathcal{V}(t)\}$ for the language accepted by the type t.

3 All-match semantics

In this section, we introduce an auxiliary definition of an all-match semantics that will be used to define our disambiguation policy and to study the problematic regular expressions. For a type t and a word w, we define

$$\texttt{M}_t(w) := \{v \in \mathcal{V}(t) \mid \exists w'.\ w = \texttt{flat}(v)@w'\}$$

This set represents all the possible ways to match a prefix of w by a value of type t. For a word w and a value $v \in \texttt{M}_t(w)$, we write $v^{-1}w$ for the (unique) word w' such that $w = \texttt{flat}(v)@w'$.

Definition 1. *A type is* problematic *if it contains a sub-expression of the form t^* where $[] \in \texttt{flat}(t)$.*

Definition 2. *A value is* problematic *if it contains a sub-value of the form $[\ldots; v; \ldots]$ with $\texttt{flat}(v) = []$. The set of non-problematic values of type t is written $\mathcal{V}^{\texttt{np}}(t)$.*

Definition 3. *A word w is* problematic *for a type t if $\texttt{M}_t(w)$ is infinite.*

The following proposition establishes the relation between these three notions.

Proposition 1. *Let t be a type. The following assertions are equivalent:*

1. *t is problematic;*
2. *there exists a problematic value in $\mathcal{V}(t)$;*
3. *there exists a word w which is problematic for t.*

We will need to do induction both on a type t and a word w. To make it formal, we introduce a well-founded ordering on pairs (t, w): $(t_1, w_1) < (t_2, w_2)$ if either t_1 is a strict syntactic sub-expression of t_2 or $t_1 = t_2$ and w_1 is a strict suffix of w_2.

We write $\mathtt{M}^{\mathtt{np}}_t(w) = \mathtt{M}_t(w) \cap \mathcal{V}^{\mathtt{np}}(t)$ for the set of non-problematic prefix matches.

Proposition 2. *The following equalities hold:*

$$\begin{array}{ll}
\mathtt{M}^{\mathtt{np}}_c(w) & = \begin{cases} \{c\} & \text{if } \exists w'.\ c :: w' = w \\ \emptyset & \text{otherwise} \end{cases} \\
\mathtt{M}^{\mathtt{np}}_{t_1 \times t_2}(w) & = \{(v_1, v_2) \mid v_1 \in \mathtt{M}^{\mathtt{np}}_{t_1}(w), v_2 \in \mathtt{M}^{\mathtt{np}}_{t_2}(v^{-1}w)\} \\
\mathtt{M}^{\mathtt{np}}_{t_1 + t_2}(w) & = \{e : v \mid e \in \{1,2\}, v \in \mathtt{M}^{\mathtt{np}}_{t_e}(w)\} \\
\mathtt{M}^{\mathtt{np}}_{t^*}(w) & = \{v :: \sigma \mid v \in \mathtt{M}^{\mathtt{np}}_t(w), \boxed{\mathtt{flat}(v) \neq []}, \sigma \in \mathtt{M}^{\mathtt{np}}_{t^*}(v^{-1}w)\} \cup \{[]\} \\
\mathtt{M}^{\mathtt{np}}_{\varepsilon}(w) & = \{\varepsilon\}
\end{array}$$

This proposition gives a naive algorithm to compute $\mathtt{M}^{\mathtt{np}}_t(w)$. Indeed, because of the condition $\mathtt{flat}(v) \neq []$ in the case for $\mathtt{M}^{\mathtt{np}}_{t^*}(w)$, the word $v^{-1}w$ is a strict suffix of w, and we can interpret the equalities as an inductive definition for the function $\mathtt{M}^{\mathtt{np}}_t(w)$ (induction on the pair (t, w)).

Note that if we remove this condition $\mathtt{flat}(v) \neq []$ and replace $\mathtt{M}^{\mathtt{np}}_{\_}(\_)$ with $\mathtt{M}_{\_}(\_)$, we get valid equalities.

Corollary 1. *For any word w and type t, $\mathtt{M}^{\mathtt{np}}_t(w)$ is finite.*

4 Disambiguation

A classical semantics of matching is defined by expanding the Kleene star t^* to $(t \times t^*) + \varepsilon$ and then relying on a disambiguation policy for the alternation (say, first-match policy). This gives a "greedy" semantics, which is sometimes meant as a local approximation of the longest match semantics. However, as described by Vansummeren [Van03], the greedy semantics does not implement the longest match policy. As a matter of fact, the greedy semantics really depends on the internals of Kleene-stars. For instance, consider the regular expressions $t_1 = ((a \times b) + a)^* \times (b + \varepsilon)$ and $t_2 = (a + (a \times b))^* \times (b + \varepsilon)$, and the word $w = ab$. With the greedy semantics, when matching w against t_1, the star captures ab, but when matching against t_2, the star captures only a.

Let t be a type. The matching problem is to compute from a word $w \in \mathtt{flat}(t)$ a value $v \in \mathcal{V}(t)$ whose flattening is w. In general, there are several different solutions. If we want to extract a single value, we need to define a disambiguation policy, that is, a way to choose a *best* value $v \in \mathcal{V}(t)$ such that $w = \texttt{flat}(v)$. Moreover, we don't want to do it by providing an algorithm, or a set of ad hoc rules. Instead, we want to give a *declarative specification* for the disambiguation policy. To do this, we introduce a total ordering on the set $\mathcal{V}(t)$, and we specify that the best value with a given flattening is the largest value for this ordering.

We define the total (lexicographic) ordering $<$ on each set $\mathcal{V}(t)$ by:

$$\begin{array}{ll}
c < c & := \texttt{false} \\
(v_1, v_2) < (v_1', v_2') & := (v_1 < v_1') \vee (v_1 = v_1' \wedge v_2 < v_2') \\
(e : v) < (e' : v') & := (e > e') \vee (e = e' \wedge v < v') \\
[] < \sigma' & := \sigma' \neq [] \\
v :: \sigma < v' :: \sigma' & := (v < v') \vee (v = v' \wedge \sigma < \sigma') \\
v :: \sigma < [] & := \texttt{false} \\
\varepsilon < \varepsilon & := \texttt{false}
\end{array}$$

This definition is well-founded by induction on the size of the values. It captures the idea of a specific disambiguation rule, namely a left-to-right policy for the sequencing, a first match policy for the alternation (we prefer the first of two alternatives, so $1 : v$ should be larger than $2 : w$) and a greedy policy for the Kleene star.

Lemma 1. *Let t be a type and v a value in $\mathcal{V}(t)$. If v is problematic, then there exists some value $v' \in \mathcal{V}(t)$ such that $\texttt{flat}(v) = \texttt{flat}(v')$ and $v' > v$.*

The idea is to consider a sub-value $\sigma = [\ldots; v_0; \ldots]$ of v with $\texttt{flat}(v_0) = []$, and to replace σ with $\sigma :: v_0$ to get v', which is stricly larger than v for the ordering. Considering this lemma and Corollary 1, it is natural to restrict our attention to non problematic values. This is meaningful, because if $w \in \texttt{flat}(t)$, then there always exist non-problematic values whose flattening is w.

Definition 4. *Let t be a type and $w \in \texttt{flat}(t)$. We define:*
$\texttt{m}_t(w) := \max_< \{v \in \mathcal{V}^{\texttt{np}}(t) \mid \texttt{flat}(v) = w\}$

The previous section gives a naive algorithm to compute $\texttt{m}_t(w)$. We can first compute the set $\texttt{M}_t^{\texttt{np}}(w)$, then filter it to keep only the values v such that $v^{-1}w = []$, and finally extract the largest value from this set (if any). This algorithm is very inefficient because it has to materialize the set $\texttt{M}_t^{\texttt{np}}(w)$, which can be very large.

The recognition algorithm in [TSY02] or [Har99] can be interpreted in terms of our ordering. It generates the set $\texttt{M}_t^{\texttt{np}}(w)$ lazily, in decreasing order, and it stops as soon as it reaches the end of the input. To do this, it uses backtracking implemented with continuations. Adapting this algorithm to the matching problem is possible, but the resulting one would be quite inefficient because of backtracking (moreover, the continuations have to hold partial values, which generates a lot of useless memory allocations).

5 A linear time matching algorithm

In this section, we present an algorithm to compute $\mathtt{m}_t(w)$ in linear time with respect to the size of w, in particular without backtracking nor useless memory allocation.

This algorithm works in two passes. The main (second) pass is driven by the syntax of the type. It builds a value from a word by induction on the type, consuming the word from the left to the right. This pass must make some choices: which branch of the alternative type $t_1 + t_2$ to consider, or how many times to iterate a Kleene star t^*. To allow making these choices without backtracking, a first preprocessing pass annotates the word with enough information.

The first pass consists in running an automaton right to left on the word, and keeping the intermediate states as annotations between each symbol of the word. The automaton is build directly on the syntax tree of the regular expression itself (its states correspond to the nodes of the regular expression syntax tree). A reviewer pointed us to a previous work [Kea91] which uses the same idea. Our presentation is more functional (hence more amenable to reasoning) and is extended to handle problematic regular expressions.

5.1 Non-problematic case

We first present an algorithm for the case when w is not problematic. Recall the following classical definition.

Definition 5. *A non-deterministic finite state automaton (FSA) with ε-transitions is a triple (Q, q_f, δ) where Q is a finite set (of states), q_f is a distinguished (final) state in Q, and $\delta \subset (Q \times \Sigma \times Q) \cup (Q \times Q)$.*

The transition relation $q_1 \xrightarrow{w} q_2$ (for $q_1, q_2 \in Q$, $w \in \Sigma^*$) is defined inductively by the following rules:

- $q_1 \xrightarrow{[]} q_2$ if $q_1 = q_2$ or $(q_1, q_2) \in \delta$
- $q_1 \xrightarrow{[c]} q_2$ if $(q_1, c, q_2) \in \delta$
- $q_1 \xrightarrow{w_1 @ w_2} q_3$ if $q_1 \xrightarrow{w_1} q_2$ and $q_2 \xrightarrow{w_2} q_3$.

We write $\mathcal{L}(q) = \{w \mid q \xrightarrow{w} q_f\}$.

From types to automata. Constructing a non-deterministic automaton from a regular expression is a standard operation. However, we need to keep a tight connection between the automata and the types. To do so, we endow the abstract syntax trees of types with a transition relation so as to turn them into automata. Formally, we introduce the set of *locations* (or nodes) $\lambda(t)$ of a type t (a location is a sequence over $\{\mathtt{fst}, \mathtt{snd}, \mathtt{lft}, \mathtt{rgt}, \mathtt{star}\}$), and for a location $l \in \lambda(t)$, we define $t.l$ as the subtree rooted at location l:

$$\begin{cases} \lambda(c) & := \{[]\} \\ \lambda(t_1 \times t_2) & := \{[]\} \cup \mathtt{fst} :: \lambda(t_1) \cup \mathtt{snd} :: \lambda(t_2) \\ \lambda(t_1 + t_2) & := \{[]\} \cup \mathtt{lft} :: \lambda(t_1) \cup \mathtt{rgt} :: \lambda(t_2) \\ \lambda(t^*) & := \{[]\} \cup \mathtt{star} :: \lambda(t) \\ \lambda(\varepsilon) & := \{[]\} \end{cases} \qquad \begin{cases} t.[] & := t \\ (t_1 \times t_2).(\mathtt{fst} :: l) & := t_1.l \\ (t_1 \times t_2).(\mathtt{snd} :: l) & := t_2.l \\ (t_1 + t_2).(\mathtt{lft} :: l) & := t_1.l \\ (t_1 + t_2).(\mathtt{rgt} :: l) & := t_2.l \\ (t^*).(\mathtt{star} :: l) & := t.l \end{cases}$$

Now, let us consider a fixed type t_0. We take: $Q := \lambda(t_0) \cup \{q_f\}$ where q_f is a fresh element.

If l is a location in t_0, the corresponding state will match all the words of the form $w_1 @ w_2$ where w_1 is matched by $t_0.l$ and w_2 is matched by the "rest" of the regular expression (Lemma 2 below gives a formal statement corresponding to this intuition).

We define the δ relation for our automaton by using the successor function $\mathtt{succ}(\_) : \lambda(t_0) \to Q$ which formalizes this notion of "rest":

$$\begin{aligned} \delta := & \{(l, c, \mathtt{succ}(l)) \mid t_0.l = c\} \\ \cup\ & \{(l, \mathtt{succ}(l)) \mid t_0.l = \varepsilon\} \\ \cup\ & \{(l, l :: \mathtt{fst}) \mid t_0.l = t_1 \times t_2\} \\ \cup\ & \{(l, l :: \mathtt{lft}), (l, l :: \mathtt{rgt}) \mid t_0.l = t_1 + t_2\} \\ \cup\ & \{(l, l :: \mathtt{star}), (l, \mathtt{succ}(l)) \mid t_0.l = t^*\} \end{aligned} \qquad \begin{cases} \mathtt{succ}([]) & := q_f \\ \mathtt{succ}(l :: \mathtt{fst}) & := l :: \mathtt{snd} \\ \mathtt{succ}(l :: \mathtt{snd}) & := \mathtt{succ}(l) \\ \mathtt{succ}(l :: \mathtt{lft}) & := \mathtt{succ}(l) \\ \mathtt{succ}(l :: \mathtt{rgt}) & := \mathtt{succ}(l) \\ \mathtt{succ}(l :: \mathtt{star}) & := l \end{cases}$$

An example for this construction will be given in the next session for the problematic case.

The following lemma relates the behavior of the automaton, the $\mathtt{succ}(\_)$ function, and the flat semantics of types.

Lemma 2. *For any location $l \in \lambda(t_0)$: $\mathcal{L}(l) = \mathtt{flat}(t_0.l) @ \mathcal{L}(\mathtt{succ}(l))$*

First pass. We can now describe the first pass of our matching algorithm. Assume that the input is $w = [c_1; \ldots; c_n]$. The algorithm computes $n+1$ sets of states $Q_0, \ldots, Q_n$ defined as $Q_i = \{q \mid q \xrightarrow{[c_{i+1}; \ldots; c_n]} q_f\}$. That is, it annotates each suffix w' of the input w by the set of states from which the final state can be reached by reading w'.

Computing the sets Q_i is easy. Indeed, consider the automaton obtained by reversing all the transitions in our automaton (Q, q_f, δ), and use it to scan w right-to-left, starting from q_f, with the classical subset construction (with forward ε-closure). Each step of the simulation corresponds to a suffix $[c_{i+1}; \ldots; c_n]$ of w, and the subset built at this step is precisely Q_i.

This pass can be done in linear time with respect to the length of w, and more precisely in time $O(|w| \times |t_0|)$ where $|w|$ is the length of w and t_0 is the size of t_0.

Second pass. The second pass is written in pseudo-ML code, as a function `build`, that takes a pair (w, l) of a word and a location $l \in \lambda(t_0)$ such that $w \in \mathcal{L}(l)$ and returns a value $v \in \mathcal{V}(t_0.l)$.

```
let build($w, l$) = (* Invariant:  $w \in \mathcal{L}(l)$  *)
 match $t_0.l$ with
 | $c$ -> $c$
 | $t_1 \times t_2$ ->
     let $v_1$ = build($w, l ::$ fst) in let $v_2$ = build($v_1^{-1}w, l ::$ snd) in $(v_1, v_2)$
 | $t_1 + t_2$ ->
     if $w \in \mathcal{L}(l ::$ lft)
     then let $v_1$ = build($w, l ::$ lft) in $1 : v_1$
     else let $v_2$ = build($w, l ::$ rgt) in $2 : v_2$
 | $t^*$ ->
     if $w \in \mathcal{L}(l ::$ star)
     then let $v$ = build($w, l ::$ star) in let $\sigma$ = build($v^{-1}w, l$) in $v :: \sigma$
     else $[]$
 | $\varepsilon$ -> $\varepsilon$
```

The following proposition explains the behavior of the algorithm, and allows us to establish its soundness.

Proposition 3. *If $w \in \mathcal{L}(l)$ and if t_0 is* **non-problematic***, then the algorithm* `build`(w, l) *returns* $\max_<\{v \in \mathcal{V}(t_0.l) \mid \exists w' \in \mathcal{L}(\texttt{succ}(l)).\ w = \texttt{flat}(v)@w'\}$.

Corollary 2. *If $w \in \texttt{flat}(t_0)$ and if t_0 is* **non-problematic***, then the algorithm* `build`$(w, [])$ *returns* $\texttt{m}_{t_0}(w)$.

Implementation. The tests $w \in \mathcal{L}(l)$ can be implemented in constant time thanks to the first pass [4] . Indeed, for a suffix w' of the input, $w' \in \mathcal{L}(l)$ means that the state l is in the set attached to w' in the first pass. Similarly, the precondition $w \in \texttt{flat}(t_0)$ can also be tested in constant time.

The second pass also runs in linear time with respect to the length of the input word (and more precisely in time $O(|w|\ |t_0|)$), because `build` is called at most once for each suffix w' of w and each location l (the number of locations is finite). This property holds because of the non-problematic assumption (otherwise the algorithm may not terminate).

Note that w is used *linearly* in the algorithm: it can be implemented as a mutable pointer on the input sequence (which is updated when the c case reads a symbol), and it doesn't need to be passed around.

5.2 Solution to the problematic case

Idea of a solution. Let us study the problem with problematic types in the algorithm from the previous section. The problem is in the case t^* of the algorithm,

[4] If the regular expressions are 1-unambiguous (which is the case for regular expressions in DTD and XML Schema [W3C01]), the tests can be implemented directly with a look-ahead of one symbol, without the first pass.

when $[] \in \texttt{flat}(t)$. Indeed, the first recursive call to `build` may return a value v such that $\texttt{flat}(v) = []$, which implies $v^{-1}w = w$, and the second recursive call has then the same arguments as the main call. In this case, the algorithm does not terminate.

This can also be seen on the automaton. If the type at location l accepts the empty sequence, there are in the automaton non-trivial paths of ε-transitions from l to l. The idea is to break these paths, by "disabling" their last transition (the one that returns to l) when no symbol has been matched in the input word since the last visit of the state l.

Here is how to do so. A location l is said to be a star node if $t_0.l = t^*$. Any sublocation l' is said to be scoped by l. Note that when the automaton starts an iteration in a star node (by using the ε transition $(l, l :: \texttt{star})$), the only way to exit the iteration (and to reach the final state) is to go back to the star node l. The idea is to prevent the automaton to enter back a star node unless some symbol has been read during the last iteration. The state of the automaton includes a flag b that is set whenever a character is read. The flag is reset when an iteration starts, that is, when a transition of the form $(l, l :: \texttt{star})$ is used. When the flag is not set, all ε transitions of the form $(l, \texttt{succ}(l))$, where $\texttt{succ}(l)$ is a star node scoping l, are disabled.

When the flag is set, this can be interpreted as the requirement: Something needs to be read in order to exit the current iteration. Consequently, it is natural to start running the automaton with the flag set, and to require the flag to be set at the final node.

From problematic types to automata. Let us make this idea formal. We write P for the set of locations l such that $\texttt{succ}(l)$ is an ancestor of l in the abstract syntax tree of t_0 (this implies that $\texttt{succ}(l)$ is a star node). Note that the "problematic" transitions are the ε-transition of the form $(l, \texttt{succ}(l))$ with $l \in P$.

We now take: $Q := (\lambda(t_0) \cup \{q_f\}) \times \{0, 1\}$. Instead of (q, b), we write q^b. The final state is q_f^1. Here is the transition relation:

$$\begin{array}{rl}
\delta_0 := & \{(l^b, c, \texttt{succ}(l)^1) \mid t_0.l = c\} \\
\cup & \{(l^b, l :: \texttt{fst}^b) \mid t_0.l = t_1 \times t_2\} \\
\cup & \{(l^b, l :: \texttt{lft}^b), (l^b, l :: \texttt{rgt}^b) \mid t_0.l = t_1 + t_2\} \\
\cup & \{(l^b, l :: \texttt{star}^0) \mid t_0.l = t^*\} \\
\cup & \{(l^b, \texttt{succ}(l)^b) \mid (*)\}
\end{array}$$

where the condition $(*)$ is the conjunction of:

(I) $t_0.l$ is either ε or a star t^*
(II) if $l \in P$, then $b = 1$

Note that the transition relation is monotonic with respect to the flag b: if $q_1^0 \xrightarrow{w} q_2^b$, then $q_1^1 \xrightarrow{w} q_2^{b'}$ for some $b' \geq b$.

We write $\mathcal{L}(q^b) := \{w \mid q^b \xrightarrow{w} q_f^1\}$. As for any FSA, we can simulate the new automaton either forwards or backwards. In particular, it is possible to annotate a word w with a right-to-left traversal (in linear time w.r.t the length of w), so as to be able to answer in constant time any question of the form $w' \in \mathcal{L}(q^b)$ where w' is a suffix of w. This can be done with the usual subset construction. The monotonicity remark above implies that whenever q^0 is in a subset, then q^1 is also in a subset, which allows to optimize the representation of the subsets.

The lemma above is the invariant used to prove Proposition 4.

Lemma 3. *Let $l \in \lambda(t_0)$ and $L = \mathtt{flat}(t_0.l)$. Then:*

$$\mathcal{L}(l^1) = L@\mathcal{L}(\mathtt{succ}(l)^1)$$
$$\mathcal{L}(l^0) = \begin{cases} (L\backslash\{[]\})@\mathcal{L}(\mathtt{succ}(l)^1) & \text{if } l \in P \vee [] \notin L \\ (L\backslash\{[]\})@\mathcal{L}(\mathtt{succ}(l)^1) \cup \mathcal{L}(\mathtt{succ}(l)^0) & \text{if } l \notin P \wedge [] \in L \end{cases}$$

Algorithm. We now give a version of the linear-time matching algorithm which supports the problematic case. The only difference is that it keeps track (in the flag b) of the fact that something has been consumed on the input since the last beginning of an iteration in a star. The first pass is not modified, except that the new automaton is used. The second pass is adapted to keep track of b.

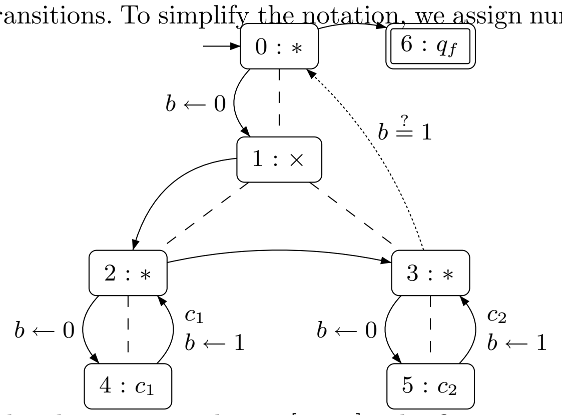
```
let build'($w, l^b$) = (* Invariant:  $w \in \mathcal{L}(l^b)$  *)
 match $t_0.l$ with
  | $c$ -> $c$
  | $t_1 \times t_2$ ->
      let $v_1$ = build'($w, l :: \mathtt{fst}^b$) in
      let $b'$ = if ($v_1^{-1}w = w$) then $b$ else 1 in
      let $v_2$ = build'($v_1^{-1}w, l :: \mathtt{snd}^{b'}$) in
      $(v_1, v_2)$
  | $t_1 + t_2$ ->
      if $w \in \mathcal{L}(l :: \mathtt{lft}^b)$
      then let $v_1$ = build'($w, l :: \mathtt{lft}^b$) in $1 : v_1$
      else let $v_2$ = build'($w, l :: \mathtt{rgt}^b$) in $2 : v_2$
  | $t^*$ ->
      if $w \in \mathcal{L}(l :: \mathtt{star}^0)$
      then let $v$ = build'($w, l :: \mathtt{star}^0$) in let $\sigma$ = build'($v^{-1}w, l^1$) in $v :: \sigma$
        (* Invariant: $v^{-1}w \neq w$ *)
      else $[]$
  | $\varepsilon$ -> $\varepsilon$
```

Proposition 4. *Let $w \in \mathcal{L}(l^b)$. Let V be the set of non-problematic values $v \in \mathcal{V}(t_0.l)$ such that $\exists w' \in \mathcal{L}(\mathtt{succ}(l)^{b'})$. $w = \mathtt{flat}(v)@w'$ with $b' = 1$ if $\mathtt{flat}(v) \neq []$ and $((b = 1 \vee l \notin P) \wedge b' = b)$ if $\mathtt{flat}(v) = []$. Then the algorithm $\mathtt{build'}(w, l^b)$ returns $\max_< V$.*

Corollary 3. *If $w \in \mathtt{flat}(t_0)$, then the algorithm $\mathtt{build}'(w, []^1)$ returns $\mathtt{m}_{t_0}(w)$.*

Implementation. The same remarks as for the first algorithm apply for this version. In particular, we can implement w and b with mutable variables which are updated in the case c (when a symbol is read); thus, we don't need to compute b' explicitly in the case $t_1 \times t_2$.

Example. To illustrate the algorithm, let us consider the problematic type $t_0 = (c_1^* \times c_2^*)^*$. The picture below represents both the syntax tree of this type (dashed lines), and the transitions of the automaton (arrows). The dotted arrow is the only problematic transition, which is disabled when $b = 0$. Transitions with no symbols are ε-transitions. To simplify the notation, we assign numbers to states.



Let us consider the input word $w = [c_2; c_1]$. The first pass of the algorithm runs the automaton backwards on this word, starting in state 6^1, and applying subset construction. In a remark above, we noticed that if i^0 is in the subset, then i^1 is also in the subset. Consequently, we write simply i to denote both states i^0, i^1. The (backward) ε-closure of 6^1 is $S_2 = \{6^1, 0^1, 3^1, 2^1, 1^1\}$. Reading the symbol c_1 from S_2 leads to the state 4, whose ε-closure is $S_1 = \{4, 2, 1, 0, 3^1\}$. Reading the symbol c_2 from S_1 leads to the state 5, whose ε-closure is $S_0 = \{5, 3, 2, 1, 0\}$.

Now we can run the algorithm on the word w with the trace $[S_0; S_1; S_2]$. The flag b is initially set. The star node 0 checks whether it must enter an iteration, that is, whether $1 \in S_0$. This is the case, so an iteration starts, and b is reset. The star node 2 returns immediately without a single iteration, because $4 \notin S_0$. But the star node 3 enters an iteration because $5 \in S_0$. This iteration consumes the first symbol of w, and sets b. After this first iteration, the current subset is S_1. As 5 is not in S_1, the iteration of the node 3 stops, and the control is given back to the star node 0. Since $1 \in S_1$, another iteration of the star 0 starts, and then similarly with an inner iteration of 2. The second symbol of w is consumed. The star node 3 (resp. 0) refuses to enter an extra iteration because $5 \notin S_2$ (resp. $1 \notin S_2$); note that $1^1 \in S_2$, but this is not enough, as this only means that an iteration could take place without consuming anything - which is precisely the situation we want to avoid.

The resulting value is $[([], [c_2]); ([c_1], [])]$. The two elements of this sequence reflect the two iterations of the star node 0.

Acknowledgments

We would like to express our gratitude to the reviewers of PLAN-X 2004 and ICALP 2004 for their comments and in particular for their bibliographical indications.

References

[BCF03a] Véronique Benzaken, Giuseppe Castagna, and Alain Frisch. ℂDuce: An XML-centric general-purpose language. In *ICFP '03*, 2003.

[BCF$^+$03b] S. Boag, D. Chamberlin, M. Fernandez, D. Florescu, J. Robie, J. Siméon, and M. Stefanescu. *XQuery 1.0: An XML Query Language*. W3C Working Draft, `http://www.w3.org/TR/xquery/`, May 2003.

[DF00] Danny Dub and Marc Feeley. Efficiently building a parse tree from a regular expression. *Acta Informatica*, 37(2):121–144, 2000.

[ECM02] ECMA. *CLI Partition I - Architecture*. `http://msdn.microsoft.com/net/ecma/`, 2002.

[Fri04] Alain Frisch. Regular tree language recognition with static information. *International Conference on Theoretical Computer Science*, 2004.

[GP03] V. Gapayev and B.C. Pierce. Regular object types. In *Proceedings of the 10th workshop FOOL*, 2003.

[Har99] Robert Harper. Proof-directed debugging. *Journal of Functional Programming*, 9(4):463–469, 1999.

[Hos01] Haruo Hosoya. *Regular Expression Types for XML*. PhD thesis, The University of Tokyo, 2001.

[Hos03] H. Hosoya. Regular expressions pattern matching: a simpler design. Unpublished manuscript, February 2003.

[HP01] Haruo Hosoya and Benjamin C. Pierce. Regular expression pattern matching for XML. In *The 25th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2001.

[HP03] Haruo Hosoya and Benjamin C. Pierce. XDuce: A typed XML processing language. *ACM Transactions on Internet Technology*, 3(2):117–148, 2003.

[HVP00] Haruo Hosoya, Jérôme Vouillon, and Benjamin C. Pierce. Regular expression types for XML. In *ICFP '00*, volume 35(9) of *SIGPLAN Notices*, 2000.

[Kea91] Steven. M. Kearns. Extending regular expressions with context operators and parse extraction. *Software - practice and experience*, 21(8):787–804, 1991.

[Lau01] Ville Laurikari. Efficient submatch addressing for regular expressions. Master's thesis, Helsinki University of Technology, 2001.

[Lev03] Michael Levin. Compiling regular patterns. In *ICFP '03*, 2003.

[MS03] Erik Meijer and Wolfram Schulte. Unifying tables, objects, and documents. In *DP-COOL 2003*, 2003.

[TSY02] Naoshi Tabuchi, Eijiro Sumii, , and Akinori Yonezawa. Regular expression types for strings in a text processing language. In *Workshop on Types in Programming (TIP)*, 2002.

[Van03] Stijn Vansummeren. Unique pattern matching in strings. Technical report, University of Limburg, 2003. `http://arXiv.org/abs/cs/0302004`.

[W3C00] W3C Recommendation. *Extensible Markup Language (XML) 1.0*, 2000.

[W3C01] W3C Recommandation. *XML Schema*, 2001.

[Xi01] Hongwei Xi. Dependent types for program termination verification. In *Logic in Computer Science*, 2001.