Greedy regular expression matching

Alain Frisch *
École Normale Supérieure

Luca Cardelli
Microsoft Research

Abstract

This paper studies the problem of matching sequences against regular expressions in order to produce structured values. More specifically, we formalize in an abstract way a greedy disambiguation policy and propose efficient matching algorithms. We also formalize and address a folklore problem of non-termination in naive implementations of the greedy semantics.

Regular expression types and patterns have been introduced in the setting of XML-oriented functional languages. Traditionnaly, all the XML values and sequences share a common uniform runtime representation. Our work suggests an alternative implementation technique, where regular expression types define not only a set of abstract flat sequences, but also a custom structured representation for such values. This paves the way to a variety of language designs and implementations to integrate XML structural types in existing languages (class-based OO languages, imperative features, constrained runtime environment, ...).

1 Introduction

1.1 Motivation

Regular expressions play a key role in XML. They are used in XML schema languages (DTD, XML-Schema, 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 of pattern matching [Lev03] and optimization of patterns in presence of static information [BCF03a], etc...

This work is a preliminary step in introducing similar ideas to imperative or object-oriented languages. One possible approach is the one pursued by the XTATIC language [GP03], which is a merger between XDUCE and C$^\#$ [ECM02a]. The value and type algebra are stratified, to allow mixing XDUCE types (regular expressions) and standard .NET CLR [ECM02b] types (classes). Concretely, there is a uniform representation of sequences, and all the XML types collapse to a single native CLR class at runtime. Because of the uniform representation, XTATIC can import subtyping from XDUCE (namely, set inclusion): the CLR does not see any of it at runtime. In order to have a type-safe implicit subtyping (that is, regular language inclusion), the XML fragments must be immutable, or some runtime checks must be introduced at runtime (as for arrays in Java or C$^\#$). Also, the uniform representation adds a lot of boxing and indirections. This can be good if the application relies a lot on subtyping, but it can hurt if the application needs fast random access on data (for instance, accessing an element in the middle of a Kleene-star requires a traversal of the sequence), want to mutate data, or needs to cooperate with non-XML parts of the application (that don't expect uniform and boxed values). Finally, collapsing all the XML types to a unique type means that type information is lost after compilation; this raises issues for separate compila-

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

tion.

The starting point of this work was to consider another approach, and study an alternative implementation technique for XTATIC. Instead of having a uniform representation of sequences, we want to represent them with native CLR constructions. In this paper, we consider *types* that are regular expressions. Unlike XDUCE, our types describe not only a set of possible sequences, but also a concrete structured representation of values. We use ×, +, *, ε to denote concatenation, alternation, Kleene star and the singleton set containing the empty sequence.

Typically, a value of type (`System.Object` × `int`) should be a struct with two members (that is, a value type in the terminology of the CLR [GS01]), whose second member is a CLR unboxed integer. Similarly, a Kleene-star type `int`* could be an array or another collection with random access features. The benefits of this approach are: (1) data is stored more compactly; (2) it can be accessed (and mutated) efficiently, and (3) it can be made compatible with non-XML specific code (and if the mapping to CLR types retains enough information from XML types, separate compilation is made possible without keeping extra information). The drawback is that we now need coercions between set-theoretic subtypes. For instance, (`int` × `int`) is a set-theoretic subtype of `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], and one might want to make it available to the programmer as well. Note that coercions cannot fail, though general pattern matching can.

Another work worth mentioning in this area is the Xen language [MS03], which adds to C$^\#$ some dose of structural types reminiscent of regular expression types. They bind structural types to native CLR types. However, they loose the nice semantic properties of subtyping and equivalences from regular expression types, and they don't have the equivalent of XDUCE/XTATIC pattern matching. Our work can also be seen as an attempt to follow the Xen approach while sticking to "pure" regular expressions types and patterns.

However, this paper does not propose a language design, such as language features or type systems. Instead, we study the theoretical problem of matching a flat sequence against a type (regular expression). The result of the process is a structured value of the given type. The algorithms we develop could be used in a variety of language designs (how to access information in a structured value; implicit or explicit coercions; mutability of values; different binding semantics for pattern matching, etc). For instance, they can directly be applied to implement the second pass of coercions between subtypes (building structured values from flat sequences).

1.2 The matching problem

The classical theory of regular expressions and automata deals mainly with the recognition problem, namely deciding whether a word belongs to the regular language described by some regular expression. In particular, two regular expressions are equivalent with respect to the recognition problem if they denote the same language.

In the matching problem, regular expressions don't only describe a language, but also a way to extract information from words in this language. This can be formalized in different ways, for instance adding capture variables to regular expressions. In this paper, we take a different approach and say that regular expressions actually denote sets of structured values. Each value can be flattened to obtain a word. For instance, the regular expression t^* denotes an array or list of values, each of type t. In particular, it is possible to access efficiently any element of the array. Similarly, the sequence regular expression $(t_1 \times t_2)$ denotes pairs (v_1, v_2).

Now, regular expression matching can be seen as the process of mapping flat words to structured values. This raises two issues:

- Semantic issue: How to deal with ambiguity in regular expressions? We can distinguish three kinds of solutions: (1) return all the possible matches, (2) disallow ambiguous regular expressions [Hos03], or (3) specify a disambiguation policy to pick a "best" result. In the setting of programming languages, we usually want to return a single result. Also, we don't want to force the programmer to rewrite regular expressions to remove ambiguities, because this process changes the structure of regular expressions and thus the structure of resulting values. Furthermore, accepting ambiguous regular expressions cannot be avoided if we want to import, say XML Schema specifications. Also, they usually lead to more compact expressions. In this paper, we focus on the point (3), and study in detail a given disambiguation policy.

- Implementation issue: How to implement matching efficiently? The classical technique of determinization allows recognition of a regular language in linear time (in the size of the word to be recognized). Can this be adapted to the matching problem?

1.3 Related work

Problematic regular expressions There is a rich literature on efficient implementation of regular expression pattern matching. For instance, Laurikari [Lau01] studies the submatch addressing problem, which extracts less information than our matching problem. Other works [Kea91, DF00] address the matching problem (referred to as "parse extraction"). A key contribution of our work is the treatment of so-called *problematic regular expressions*, together with a clean formalization of a specific disambiguation policy.

Indeed, there is a folklore problem with expression-based implementations of regular expression matching (as opposed to purely automaton-based approaches, which are not suitable for the matching problem): 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. Actually, this could even be a problem for *defining* the disambiguation policy when it is only given by a matching algorithm.

Harper [Har99] and Kearns [Kea91] (who speaks of "horrible possibility" for the non-termination problem) 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 regular expression can explode in the rewriting. Moreover, this solution has two drawbacks when we consider the matching problem:

- Changing the regular expression changes the type of the resulting values. A value of type t' is not a value of type t.
- The interaction with the disambiguation policy (see below) is not trivial. In particular, we don't see any way to design a rewriting strategy that preserves the disambiguation policy.

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. 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.

Disambiguation policy Specifying a disambiguation policy can be done by providing an explicit matching algorithm. For instance, Vansummeren [Van03] axiomatizes a longest match semantics for the Kleene star with a formal system describing the matching relation. This semantics has a "global" flavor, in the sense that the part of the word matched by a Kleene star t^* depends only on the language accepted by t and the context of the star, not its internal structure.

A more classical semantics 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.

Because of its local nature, the greedy semantics seems easier to implement, and maybe to understand (this point is questionable). Moreover, it has actually been used as the disambiguation policy in several programming languages (XDUCE, ℂDuce, λ^{re} [TSY02]), and at least for this reason it deserves attention.

In this paper, we formalize the greedy semantics by a specification that is independent of any concrete matching algorithm. We define a total ordering on values and specify that the largest possible value must be extracted. The disambiguation policy is then formalized as an optimization problem (extract the largest value with the given flattening). This is similar to the formalization of XDuce pattern matching relation [Hos01, section 2.4.2], except that we tackle with the difficulty of problematic expressions which are rejected in [Hos01].

Implementation of matching A naive backtracking implementation of the greedy semantics is quite easy to give (for the recognition problem, see for instance [TSY02], or [Har99] for the ungreedy variant). In this paper, we provide a linear time algorithm that works in two passes. The idea is to use a first pass to annotate the word and avoid backtracking in the second pass, when the value is constructed.

The first pass scans the word by running a finite state automaton. 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.

The second pass follows closely the syntax of the regular expression, and is thus very flexible. For instance, one can easily add actions to each node of the regular expression, extract only relevant information, or in the setting of a compiler, generate specialized code for a given regular expression.

2 Notations

Sequences For any set X, we write X^* for the set of sequences over X. Such a sequence is written $[x_1; \ldots; x_n]$. 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: $x :: X_1 = \{x :: s \mid s \in X_1\}$, $X_1 @ X_2 = \{s_1 @ s_2 \mid s_i \in X_i\}$.

Symbols, words We assume to be given a fixed alphabet Σ, whose elements are called symbols (they will be denoted with $c, c_1, \ldots$). 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:

$$\begin{array}{lcl} \mathcal{V}(c) & := & \{c\} \\ \mathcal{V}(t_1 \times t_2) & := & \mathcal{V}(t_1) \times \mathcal{V}(t_2) \\ \mathcal{V}(t_1 + t_2) & := & \mathcal{V}(t_1) + \mathcal{V}(t_2) \\ \mathcal{V}(t^*) & := & \mathcal{V}(t)^* \\ \mathcal{V}(\varepsilon) & := & \{\varepsilon\} \end{array}$$

On the right-hand side of this definition, × denotes the usual Cartesian product, and + the disjoint union. A value of type $t_1 \times t_2$ is written (v_1, v_2) (with $v_i \in \mathcal{V}(t_i)$). A value of type $t_1 + t_2$ is written $e : v$ (with $e \in \{1, 2\}$ and $v \in \mathcal{V}(t_i)$). Elements of $\mathcal{V}(t^*)$ can be seen as lists or arrays of values of type t; we will use the letter σ to denote them. 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:

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

We write $\mathcal{T}(t) = \{\texttt{flat}(v) \mid v \in \mathcal{V}(t)\}$ 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 $w \in \texttt{flat}(t)$, 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 way 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 w/v 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 $[] \in \mathcal{T}(t)$.*

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 establish 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 often 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 sub-expression of t_2 or $t_1 = t_2$ and w_1 is a strict suffix of w_2.

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

Proposition 2 *The following equalities hold:*

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

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

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

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

4 Disambiguation

Let t be a type. The matching problem is to compute from a word $w \in \mathcal{T}(t)$ 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 $w = \mathtt{flat}(v)$. 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.

A first step is to reject problematic values. This is meaningful, because if $w \in \mathcal{T}(t)$, then there always exist non-problematic values whose flattening is w. Moreover, if there is a problematic value whose flattening is w, then there are an infinite number of such values. Since we want to specify the best value as being the largest one for a specific ordering (see below), having an infinite number of them is problematic.

Now we need to choose amongst the remaining non-problematic values. 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 *non-problematic* value for this order.

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

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

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 and a greedy policy for the Kleene star.

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

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

The recognition algorithm in [TSY02] or [Har99] can be interpreted in terms of our ordering. It generates the set $\mathtt{M}_t^{\mathtt{np}}(w)$ 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 continuation 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 preprocessing 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.

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 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 nondeterministic 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 define a structure of automaton directly on types seen as abstract syntax trees. Formally, we introduce the set of *locations* (or nodes) $\lambda(t)$ of a type t (a location is a sequence over $\{\texttt{fst}, \texttt{snd}, \texttt{lft}, \texttt{rgt}, \texttt{star}\}$):

$$\begin{array}{ll}
\lambda(c) & := \{[]\} \\
\lambda(t_1 \times t_2) & := \{[]\} \cup \texttt{fst} :: \lambda(t_1) \cup \texttt{snd} :: \lambda(t_2) \\
\lambda(t_1 + t_2) & := \{[]\} \cup \texttt{lft} :: \lambda(t_1) \cup \texttt{rgt} :: \lambda(t_2) \\
\lambda(t^*) & := \{[]\} \cup \texttt{star} :: \lambda(t) \\
\lambda(\varepsilon) & := \{[]\}
\end{array}$$

For a type t and a location $l \in \lambda(t)$, we define $t.l$ as the subtree rooted at location l:

$$\begin{array}{lcl}
t.[] & := & t \\
(t_1 \times t_2).(\texttt{fst} :: l) & := & t_1.l \\
(t_1 \times t_2).(\texttt{snd} :: l) & := & t_2.l \\
(t_1 + t_2).(\texttt{lft} :: l) & := & t_1.l \\
(t_1 + t_2).(\texttt{rgt} :: l) & := & t_2.l \\
(t^*).(\texttt{star} :: l) & := & t.l
\end{array}$$

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 1 below gives a formal statement corresponding to this intuition).

This notion of "rest" is formalized by the successor function $\lambda(t_0) \to Q$.

$$\begin{array}{lcl}
\texttt{succ}([]) & := & q_f \\
\texttt{succ}(l :: \texttt{fst}) & := & l :: \texttt{snd} \\
\texttt{succ}(l :: \texttt{snd}) & := & \texttt{succ}(l) \\
\texttt{succ}(l :: \texttt{lft}) & := & \texttt{succ}(l) \\
\texttt{succ}(l :: \texttt{rgt}) & := & \texttt{succ}(l) \\
\texttt{succ}(l :: \texttt{star}) & := & l
\end{array}$$

We now define the δ relation for our automaton:

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

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 $\texttt{succ}(\_)$ function, and the flat semantics of types.

Lemma 1 *For any location $l \in \lambda(t_0)$: $\mathcal{L}(l) = \mathcal{T}(t_0.l)@\mathcal{L}(\texttt{succ}(l))$*

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 $Q_n, \ldots, Q_0$ 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 a time $O(|w| \; |t_0|)$ 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 $l \in \lambda(t_0)$ 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($w/v_1, l$ :: 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($w/v, l$) 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* $w \in \mathcal{T}(t_0)$ *and if* t_0 *is* **non-problematic**, *then the algorithm* build$(w, [])$ *returns* $\mathtt{m}_{t_0}(w)$.

Implementation The tests $w \in \mathcal{L}(l)$ can be implemented in constant time thanks to the first pass. 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 $w \in \mathcal{T}(t_0)$ 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, when $[] \in \mathcal{T}(t)$. Indeed, the first recursive call to build may return a value v such that $\texttt{flat}(v) = []$, which implies $w/v = w$, 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. This can be done by disabling the ε-transitions of the form $(l, \texttt{succ}(l))$, where $\texttt{succ}(l)$ is a star node scoping l. Concretely, the automaton keeps track of its current state plus a flag b that remembers if something has been read since the last beginning of an iteration in a star.

When a symbol is read, that is, when a transition of the form (l, c, l') is used, the flag is set. When an iteration starts, that is, when a transition $(l, l :: \texttt{star})$ is used, the automaton reset the flag. Then, we just need to disable the ε-transitions $(l, \texttt{succ}(l))$ where $\texttt{succ}(l)$ is a star node that scopes l when the flag is not set. The flag can then 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 $q_1^0 \xrightarrow{w} q_2^{b'}$ 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.

For a set X and a condition C, we write $\mathbf{1}_{[C]}(X)$ to denote X when C holds, and $\emptyset$ otherwise.

Lemma 2 *Let $l \in \lambda(t_0)$ and $L = \mathcal{T}(t_0.l)$. Then:*

$$\begin{array}{rcl} \mathcal{L}(l^1) & = & L @ \mathcal{L}(\texttt{succ}(l)^1) \\ \mathcal{L}(l^0) & = & (L \backslash \{[]\}) @ \mathcal{L}(\texttt{succ}(l)^1) \\ & \cup & \mathbf{1}_{[l \notin P \wedge [] \in L]}(\mathcal{L}(\texttt{succ}(l)^0)) \end{array}$$

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 be 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.

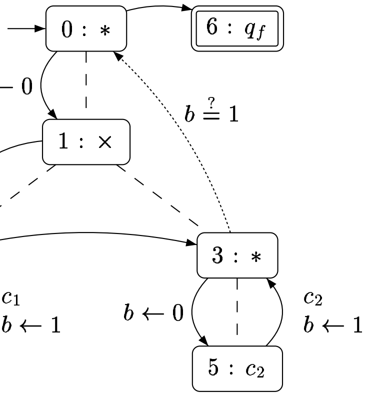
```
let build'$(w, l^b)$ =
  (* Invariant:  $w \in \mathcal{L}(l^b)$  *)
  match $t_0.l$ with
  | $c$ -> $c$
  | $t_1 \times t_2$ ->
       let $v_1$ = build'$(w, l :: \texttt{fst}^b)$ in
       let $b'$ = if $(w/v_1 = w)$ then $b$ else 1 in
       let $v_2$ = build'$(w/v_1, l :: \texttt{snd}^{b'})$ in
       $(v_1, v_2)$
  | $t_1 + t_2$ ->
       if $w \in \mathcal{L}(l :: \texttt{lft}^b)$ then
         let $v_1$ = build'$(w, l :: \texttt{lft}^b)$ in
         $1 : v_1$
       else
         let $v_2$ = build'$(w, l :: \texttt{rgt}^b)$ in
         $2 : v_2$
  | $t^*$ ->
       if $w \in \mathcal{L}(l :: \texttt{star}^0)$ then
         let $v$ = build'$(w, l :: \texttt{star}^0)$ in
         (* Invariant: $w/v \neq w$ *)
         let $\sigma$ = build'$(w/v, l^1)$ in
         $v :: \sigma$
       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 $\exists w' \in \mathcal{L}(\texttt{succ}(l)^{b'})$. $w = \texttt{flat}(v) @ w'$ with $b' = 1$ if $\texttt{flat}(v) \neq []$ and $((b = 1 \vee l \notin P) \wedge b' = b)$ if $\texttt{flat}(v) = []$. Then the algorithm* $\texttt{build}'(w, l^b)$ *returns* $\max_< V$.

Corollary 3 *If $w \in \mathcal{T}(t_0)$, then the algorithm* $\texttt{build}'(w, []^1)$ *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 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 $[([], [c2]); ([c1], [])]$. The two elements of this sequence reflect the two iterations of the star node 0.

6 Extensions, variants

More regular expressions We have presented a limited set of regular expression constructors. We could easily extend all our definitions and results, to include for instance:

- Kleene-star with ungreedy-semantics: for this constructor, the empty sequence is the largest value instead of being the smallest one, in the disambiguation ordering. The corresponding case in the matching algorithm simply tries to return [] when possible, instead of trying to make an extra iteration. Note that the two kinds of Kleene-star could easily cohabit in our framework.

- Non-empty iteration operators t^+, with two variants (greedy and ungreedy).

- Right-context sensitivity operator: $\%t$ that “matches” t but do not remove the corresponding subsequence from the input sequence.

It would be possible to adapt our formalism to capture only interesting parts of sequences by introducing explicit capture variables.

Whether our technique can be adapted to deal with the longest match semantics [Van03] is an open question.

Language design We presented here regular expressions over a finite set of symbols. In a real language design with named typing, we could imagine that regular expressions are built on top of named types, plus some singleton values (character singletons, for instance).

In the design of a language, we can imagine that the programmer could provide custom types to implement containers for regular expression types (this would allow the programmer to use pre-existing types of the language). For instance, in the setting of an extension to C$^{\#}$, the programmer could match a sequence against an existing struct type with public fields, or a class type (the pattern matcher would then call the constructor of this class to store the result). In the setting of an XML-oriented language, there would probably be a specific type `PCDATA`, equivalent to `char*` as for the denoted languages, but with a compact string representation (for instance `System.String`). The idea here is that our matching algorithm allows the implementation to choose custom concrete representation for values.

Our algorithm can also be adapted to add to an existing language (*without* regular expression types) some kind of regular expression pattern matching of sequences. For instance, we have implemented as an extension to a C$^{\#}$ compiler a construction that matches an array of type `object[]` against a regular expression built from C$^{\#}$ types; it is possible to bind identifiers to elements of the array and to perform arbitrary operations (C$^{\#}$statement) at each node of the regular expression.

Type inference In this work we don't address the question of type inference, that is computing for each node of a type the regular language of all substrings that can be matched by that node (with the given disambiguation policy), when the input is restricted to belong to some given regular language. We believe that an algorithm for computing these regular languages could be derived from our matching algorithm, by applying it symbolically to a whole regular language instead of a single input word.

Optimizations We presented our algorithm as a two passes process. The first one scans completely the input word. In some cases, this can be avoided. For instance, if one knows statically that the flat sequence is matched by the type (for instance because the flat sequence was obtained from flattening a structured value whose type is known at compile time), we can start running the main algorithm, and only when some test is needed, we run the automaton (on the current suffix of the input). In some case, a bounded look-ahead on the sequence can completely avoid the scan. This is the case if the automaton associated to the type has the so-called Glushkov deterministic property, namely that looking at the next symbol of the input removes non determinism. In particular, this is the case with regular expressions in DTD and in XML Schema specifications.

Acknowledgments

We would like to express our gratitude to the anonymous reviewers of PLAN-X 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.

[ECM02a] ECMA. *C# Language Specification*. `http://msdn.microsoft.com/net/ecma/`, 2002.

[ECM02b] ECMA. *CLI Partition I - Architecture*. `http://msdn.microsoft.com/net/ecma/`, 2002.

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

[GS01] Andrew D. Gordon and Don Syme. Typing a multi-language intermediate code. *ACM SIGPLAN Notices*, 36(3):248–260, 2001.

[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`.

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