Regular tree language recognition with static information

Alain Frisch
Département d'Informatique
École Normale Supérieure, Paris, France
Alain.Frisch@ens.fr

Abstract

This paper presents our compilation strategy to produce efficient code for pattern matching in the CDuce compiler, taking into account static information provided by the type system. Indeed, this information allows in many cases to compute the result (that is, to decide which branch to consider) by looking only at a small fragment of the tree. Formally, we introduce a new kind of deterministic tree automata that can efficiently recognize regular tree languages with static information about the trees and we propose a compilation algorithm to produce these automata.

1 Introduction

Emergence of XML has given tree automata theory a renewed importance[Nev02]. Indeed, XML schema languages such as DTD, XML-Schema, Relax-NG describe more or less regular languages of XML documents (considered as trees). Consequently, recent XML-oriented typed programming languages such as XDuce [Hos00, HP02], CDuce [BCF03, FCB02], Xtatic [GP03] have type algebras where types denote regular tree languages. The type system of these languages relies on a subtyping relation, defined as the inclusion of the regular languages denoted by types, which is known to be a decidable problem (new and efficient algorithms have been designed for this purpose, and they behave well in practice).

An essential ingredient of these languages is a powerful pattern matching operation. A pattern is a declarative way to extract information from an XML tree. Because of this declarative nature, language implementors have to propose efficient execution models for pattern matching.

This paper describes our approach in implementing pattern matching in CDuce[1]. To simplify the presentation, the paper studies only a restricted form of pattern matching, without capture variable and with a very simple kind of trees. Of course, our implementation handles capture variables and the full set of types and patterns constructors in CDuce. In the simplified form, the pattern matching problem is a recognition problem, namely deciding whether a tree v belongs to a regular tree language X or not.

1. If the regular language is given by a tree automaton, a top-down recognition algorithm may have to backtrack, and the recognition time is not linear in the size of the input tree.

2. It is well-known that any tree automaton can be transformed into an equivalent bottom-up deterministic automaton, which ensures linear execution time; the size of the automaton may be huge even for simple languages, which can make this approach unfeasible in practice.

3. The static type system of the language provides an upper approximation for the type of the matched tree v, that is some regular language X_0 such that v is necessarily in X_0. Taking this information into account, it should be possible to avoid looking at some subtree of v. However, classical bottom-up tree automata are bound to look at the whole tree, and they cannot take this kind of static knowledge into account.

Let us give an example to illustrate the last point. Consider the following CDuce program:

[1]CDuce is available for download at http://www.cduce.org/.

```
type A = <a>[ A* ]
type B = <b>[ B* ]

let f ((A|B)->Int) A    ->0 | B ->1
let g ((A|B)->Int) <a>_->0 | _ ->1
```

The first lines introduce two types A and B. They denote XML documents with only <a> (resp. <b>) tags and nothing else. Then two functions f and g are defined. Both functions take an argument which is either a document of type A or of type B. They return 1 when the argument is of type A, and 0 when the argument is of type B. The declaration of g suggests an efficient execution schema: one just has to look at the root tag to answer the question. Instead, if we consider only the body of f, we have to look at the whole argument, and check that every node of the argument is tagged with <a> (resp. with <b>); whatever technique we use - deterministic bottom-up or backtracking top-down - it will be less efficient than g.

But if we use the information given by the function interface, we know that the argument is necessarily of type A or of type B, and we can compile f exactly as we compile g.

This example demonstrates that taking static information into account is crucial to provide efficient execution for declarative patterns as in f.

The main contribution of this paper is the definition of a new kind of deterministic bottom-up tree automata, called NUA (non-uniform automata) and a compilation algorithm that produces an efficient NUA equivalent to a given non-deterministic (classical) automaton, taking into account static knowledge about the matched trees.

Informally, non-uniform automata enrich classical bottom-up automata with a “control state”. The control state is threaded through the tree, during a top-down and left-to-right traversal. In some cases, it is possible to stop the traversal of a whole subtree only by looking at the current state. Non-uniform automata combine the advantage of deterministic bottom-up and deterministic top-down tree automata, and they can take benefit from static information.

In order to discriminate several regular tree languages, a NUA does not necessarily need to consider a given subtree at all. The “magic” of the compilation algorithm is to compute a NUA that will extract only a *minimal* set of information from trees. More precisely, a NUA will skip a subtree if and only if looking at this tree does not bring any additional relevant information, considering the initial static information and the information already gathered during the beginning of the run.

Remark 1.1 *A central idea in XDuce-like languages is that XML document live in an untyped world and that XML types are structural. This is in contrast with the XML Schema philosophy, whose data model (after validation) attaches type* names *to XML nodes. Moreover, in XML Schema, the context and the tag of an element are enough to know the exact XML Schema type of the element. In XDuce-like languages, in general, one may have to look deep inside the elements to check type constraints. Our work shows how an efficient compilation of pattern matching can avoid this costly checks: our compilation algorithm detects when the context and the tag are enough to decide of the type of an element without looking at its content. This work supports the claim that a structural data model à la XDuce can be implemented as efficiently as a data model with explicit type names à la XML Schema.*

Related work Levin [Lev03] also addresses the implementation of pattern matching in XDuce-like programming languages. He introduces a general framework (intermediate language, matching automata) to reason about the compilation of patterns, and he proposes several compilation strategies. He leaves apart the issue of using static types for compilation, which is the main motivation for our work. So the two works are complementary: our compilation algorithm could probably be re-casted in his formalism.

Neumann and Seidl [NS98] introduce push-down automata to locate efficiently nodes in an XML tree. Our automata shares with push-down automata the idea of threading a control-state through the tree. The formalisms are quite different because we work with simpler kind of automata (binary trees with labeled leaves, whereas they have unranked labeled forests), and we explicitly distinguish betwen control states (threaded through the tree) and results (used in particular to update the state). However, using an encoding of unranked trees in binary trees, we believe that the two notions of automata are isomorphic. But again, they don't address the issue of using static information to improve the automata, which is our main technical contribution. It should be possible to adapt our compilation algorithm to their push-down automata setting, but it would probably result in an extremely complex technical presentation. This motivates us working with simpler kinds of tree and automata.

2 Technical framework

In this section, we introduce our technical framework. We consider the simplest form of trees: binary trees with labeled leafs and unlabeled nodes. Any kind of ordered trees (n-ary, ranked, unranked; with or without labeled nodes) can be *encoded*, and the notion of regular language is invariant under these encodings. Using this very simply kind of trees simplifies the presentation.

2.1 Trees and classical tree automata

Definition 2.1 *Let Σ be a (fixed) finite set of symbols. A tree v is either a symbol $a \in \Sigma$ or a pair of trees (v_1, v_2). The set of trees is written $\mathscr{V}$.*

CDuce actually uses this form of trees to represent XML documents: forgetting about XML attributes, an XML element is represented as a pair $(tag, content)$ where tag is a leaf representing the tag and $content$ is the encoding of the sequence of children. The empty sequences is encoded as a leaf nil, and a non-empty sequence is encoded as a pair $(head, tail)$. We recall the classical definition of a tree automaton, adapted to our definition of trees.

Definition 2.2 (Tree automaton) *A* (non-deterministic) tree automaton (NDTA) *is a pair $\mathscr{A} = (R, \delta)$ where R is a finite set of* nodes*, and $\delta \subseteq (\Sigma \times R) \cup (R \times R \times R)$.*

Each node r in a NDTA defines a subset $\mathscr{A}[\![r]\!]$ of $\mathscr{V}$. These sets can be defined by the following mutually recursive equations:

$$\mathscr{A}[\![r]\!] = \{a \in \Sigma \mid (a, r) \in \delta\} \cup \bigcup_{(r_1, r_2, r) \in \delta} \mathscr{A}[\![r_1]\!] \times \mathscr{A}[\![r_2]\!]$$

We write $\mathscr{A}[\![r]\!]^2 = \mathscr{A}[\![r]\!] \cap \mathscr{V} \times \mathscr{V}$. By definition, a regular language is a subset of the form $\mathscr{A}[\![r]\!]$ for some NDTA $\mathscr{A}$ and some node r. We say that this language is *defined* by $\mathscr{A}$. There are two classical notions of deterministic tree automata:

- Top-down deterministic automata (TDDTA) satisfy the property: $\{(r_1, r_2) \mid (r_1, r_2, r) \in \delta\}$ has at most one element for any node r. These automata are strictly weaker than NDTA in terms of expressive power (they cannot define all the regular languages).
- Bottom-up deterministic automata (DTA) satisfy the property: $\{r \mid (r_1, r_2, r) \in \delta\}$ has at most one element for any pair of nodes (r_1, r_2), and similarly for the sets $\{r \mid (a, r) \in \delta\}$ with $a \in \Sigma$. These automata have the same expressive power as NDTA.

Remark 2.3 *We use the non-standard terminology of nodes instead of states. The reason is that we are going to split this notion in two: results and control states. Results will correspond to nodes in a DTA, and control states will correspond to nodes in TDDTA.*

In order to motivate the use of a different kind of automata, let us introduce different notions of context. During the traversal of a tree, an automaton computes and gathers information. The amount of extracted information can only depend on the context of the current location in the tree. A top-down recognizer (for TDDTA) can only propagate information downwards: the context of a location is thus the path from the root to the location (“upward context”). A bottom-up recognizer propagates information upwards: the context is the whole subtree rooted at the current location (“downward context”).

Top-down algorithms are more efficient when the relevant information is located near the root. For instance, going back to the CDuce examples in the introduction, we see easily that the function g should be implemented by starting the traversal from the root of the tree, since looking only at the root tag is enough (note that because of the encoding of XML documents in CDuce, the root tag is actually the left child of the root). Patterns in CDuce tend to look in priority near the root of the trees instead of their leafs. However, because of their lack of expressive power, pure TDDTA cannot be used in general. Also, since they perform independant computations of the left and the right children of a location in a tree, they cannot use information gathered in the left subtree to guide the computation in the right subtree.

The idea behind push-down automata is to traverse each node twice. A location is first entered in a given context, some computation is performed on the subtree, and the location is entered again with a new context. When a location is first entered, the context is the path from the root, but also all the “left siblings” of these locations and their subtrees (we call this the “up/left context” of the location). After the computation on the children, the context also include the subtree. The notion of non-uniform automata we are going to introduce is a slight variation on this idea: a location is entered three times. Indeed, when computing on a tree which is a pair, the automaton considers the left and right subtree in sequence. Between the two, the location is entered again to update its context, and possibly use the information gathered on the left subtree to guide the computation on the right subtree.

This richer notion of context allows to combine the advantages of DTA and TDDTA, and more.

Due to lack of space, we cannot give more background information about regular language and tree automata. To understand the technical details that follow, some familiarity with the theory of tree automata is expected (see for instance [Nev02] for an introduction to automata theory for XML and relevant bibliographical references). However, we give enough intuition so that the reader not familiar with this theory can (hopefully) grasps the main ideas.

2.2 Non-uniform automata

We now introduce a new kind of tree automata: non-uniform automata (NUA in short). They can be seen as (a generalization of) a merger between DTA and TDDTA. Let us call “results” (resp. “control state”) the nodes of DTA (resp. TDDTA). We are going to use these two notions in parallel. A current “control state” is threaded and updated during a depth-first left-to-right traversal of the tree (this control generalizes the one of TDDTA, where the state is only propagated downwards), and each control state q has its own set of results $R(q)$. Of course, the transition relation is parametric in q.

Remark 2.4 *We can see these automata as deterministic bottom-up automata enriched with a control state that makes their behavior change according to the current location in the tree (hence the terminology “non-uniform”).*

When the automaton is facing a tree (v_1, v_2) in a state q, it starts with some computation on v_1 using a new state $q_1 = \texttt{left}(q)$ computed from the current one, as for a TDDTA. This gives a result r_1 which is immediately used to compute the state $q_2 = \texttt{right}(q, r_1)$. Note that contrary to TDDTA, q_2 depends not only on q, but also on the computation performed on the left subtree. The computation on v_2 is done from this state q_2, and it returns a result r_2. As for classical bottom-up deterministic automata, the result for (v_1, v_2) is then computed from r_1 and r_2 (and q).

Let us formalize the definition of non-uniform automata. We define only the deterministic version.

Definition 2.5 *A non-uniform automaton $\mathscr{A}$ is given by a finite set of states Q, and for each state $q \in Q$:*

- *A finite set of results $R(q)$.*
- *A state $\texttt{left}(q) \in Q$.*
- *For any result $r_1 \in R(\texttt{left}(q))$, a state $\texttt{right}(q, r_1) \in Q$.*
- *For any result $r_1 \in R(\texttt{left}(q))$, and any result $r_2 \in R(\texttt{right}(q, r_1))$, a result $\delta^2(q, r_1, r_2) \in R(q)$.*
- *A* partial *function $\delta^0(q, \_) : \Sigma \to R(q)$.*

The *result* of the automaton from a state q on an input $v \in \mathscr{V}$, written $\mathscr{A}(q, v)$, is the element of $R(q)$ defined by induction on v:

$$\begin{array}{lcl} \mathscr{A}(q, a) & = & \delta^0(q, a) \\ \mathscr{A}(q, (v_1, v_2)) & = & \delta^2(q, r_1, r_2) \\ \quad \text{where} & & r_1 = \mathscr{A}(\texttt{left}(q), v_1) \\ & & r_2 = \mathscr{A}(\texttt{right}(q, r_1), v_2) \end{array}$$

Because the functions $\delta^0(q, \_)$ are partial, so are the $\mathscr{A}(q, \_)$. We write $\texttt{Dom}(q)$ the set of trees v such that $\mathscr{A}(q, v)$ is defined.

Remark 2.6 *Note that this definition boils down to that of a DTA when Q is a singleton $\{q\}$. The set of results of the NUA (for the only state) corresponds to the set of nodes of the DTA.*

It is also possible to convert a TDDTA to a NUA of the same size. The set of states of the NYA corresponds to the *set of nodes of the TDDTA, and all the states have a single result.*

Our definition of NUAs (and more generally, the class of push down automata [NS98]) is flexible enough to simulate DTA and TDDTA (without explosion of size). They allow to merge the benefit of both kind of deterministic automata, and more (neither DTA nor TDDTA can thread information from a left subtree to the right subtree).

Neumann and Seidl [NS98] introduce a family of regular languages to demonstrate that their push-down automata are exponentially more succint than deterministic bottom-up automata. The same kind of example applies for NUAs.

A pair (q, r) with $q \in Q$ and $r \in R(q)$ is called a *state-result* pair. For such a pair, we write $\mathscr{A}[\![q;r]\!] = \{v \mid \mathscr{A}(q,v) = r\}$ for the set of trees yielding result r starting from initial state q. The reader is invited to check that a NUA can be interpreted as a non-deterministic tree automata whose nodes are state-result pairs. Consequently, the expressive power of NUAs (that is the class of languages of the form $\mathscr{A}[\![q;r]\!]$) is the same as NDTAs (ie: they can define only regular languages). The point is that the definition of NUAs gives an efficient execution strategy.

Running a NUA The definition of $\mathscr{A}(q,v)$ defines an effective algorithm that operates in linear time with respect to the size of v. We will only run this algorithm for trees v which are known *a priori* to be in $\mathtt{Dom}(q)$. This is because of the intended use of the theory (compilation of CDuce pattern matching): indeed, the static type system in CDuce ensures exhaustivity of pattern matching.

An important remark: the flexibility of having a different set of results for each state makes it possible to short-cut the inductive definition and completely ignore subtrees. Indeed, as soon as the algorithm reaches a subtree v' in a state q' such that $R(q')$ is a singleton, it can directly returns without even looking at v'.

Reduction A first criterion for a NUA to be good is the following condition:

Definition 2.7 *A NUA $\mathscr{A}$ is reduced if:*

$$\forall q \in Q.\ \forall r \in R(q).\ \exists v \in \mathtt{Dom}(q).\ \mathscr{A}(q,v) = r$$

This condition means that any result of any state can be reached for some tree in the domain of the state. Since the NUA will skip a subtree if and only if the set $R(q)$ is a singleton, it is important to enforce this property.

Of course, the set of reachable results can be computed (this amounts to checking emptiness of states of a NDTA, which can be done in linear time), and so we can remove unreachable results.

The point is that we are going to present a top-down compilation algorithm (it defines a NUA by giving explicitly for each state q the set of results $R(q)$ and the transition functions, see Section 3.8). Hence the produced NUA does not need to be fully built at compile time. Consequently, it is meaningful to say that this construction directly yields a reduced NUA, and does not require to fully materialize the automaton in order to remove unreachable results.

Tail-recursion Running a NUA on a tree requires a size of stack proportional to the height of the tree. This is problematic when dealing with trees obtained by a translation from, say, huge XML trees to binary trees. Indeed, this translation transforms long sequences to deep trees, strongly balanced to the right.

The following definition will help us in these cases.

Definition 2.8 *A NUA is* tail-recursive *if, for any state $q \in Q$:*

$$\forall r_1 \in R(\mathtt{left}(q)).\forall r_2 \in R(\mathtt{right}(q, r_1)).$$
$$\delta^2(q, r_1, r_2) = r_2$$

The idea is that a tail-recursive NUA can be implemented with a tail-recursive call on the right subtree. The stack-size used by a run of the NUA is then proportional to the largest number of "left" edges on an arbitrary branch of the tree.

The theorem above shows how to turn an arbitrary NUA into a tail-recursive one.

Theorem 2.9 *Let $\mathscr{A}$ be an arbitrary NUA, and $\mathscr{A}'$ be the NUA defined by:*

$$\begin{array}{l}
Q' = \{(q, q', \sigma) \mid q, q' \in Q, \sigma : R(q) \to R(q')\} \\
R((q, q', \sigma)) = \sigma(R(q)) \\
\mathtt{left}((q, q', \sigma)) = (\mathtt{left}(q), q, \mathit{Id}) \\
\mathtt{right}((q, q', \sigma), r_1) = (\mathtt{right}(q, r_1), q', \sigma \circ \delta^2(q, r_1, \_)) \\
\delta^2((q, q', \sigma), r_1, r_2) = r_2 \\
\delta^0((q, q', \sigma), a) = \sigma \circ \delta^0(q, a)
\end{array}$$

Then:

- *$\mathscr{A}'$ is tail-recursive*
- *For any tree v: $\mathscr{A}'((q,q',\sigma),v) = \sigma \circ \mathscr{A}(q,v)$*
- *If $\mathscr{A}$ is reduced, then $\mathscr{A}'$ is also reduced.*

When the original automaton enters the right subtree, the set of results changes: a result returned by the computation on the right subtree will have to be translated to make it compatible with the set of result for the current location.

The idea is to push this translation in the computation on the right subtree. This is done by encoding the translation in the control state passed to the right subtree. In a triple (q,q',σ), the real "control-state" is q and σ encodes the translation from results of q to result of q'. This means that if (q,q',σ) is the current control state of the new NUA, then q would be the control state of the original NUA at the same point of the traversal, and σ would be the translation to be applied to the result by ancestors.

Remark 2.10 *If for some state q, any $R(\texttt{right}(q,r_1))$ is a singleton $\{\sigma'(r_1)\}$, it is possible to implement this state with a tail recursive-call on the left-subtree; in the construction above, we would take: $\texttt{left}(q,q',\sigma) = (\texttt{left}(q),q',\sigma\circ\sigma')$*

3 The algorithm

Different NUA can perform the same computation with different complexities (that is, they can ignore more or fewer subtrees of the input). To obtain efficient NUA, the objective is to keep the set of results $R(q)$ as small as possible, because when $R(q)$ is a singleton, we can drop the corresponding subtree.

Also, we want to build NUAs that take static information about the input trees into account. Hopefully, we have the opportunity of defining *partial* states, whose domain is not the whole set of trees.

In this section, we present an algorithm to build an efficient NUA to solve the dispatch problem under static knowledge. Namely, given $n+1$ regular languages $X_0,\dots,X_n$, we want to compute efficiently for any tree v in X_0 the set $\{i = 1..n \mid v \in X_i\}$.

3.1 Intuitions

Let us consider four regular languages X_1, X_2, X_3, X_4, and let $X = (X_1 \times X_2) \cup (X_3 \times X_4)$. Imagine we want to recognize the language X without static information ($X_0 = \mathscr{V}$). If we are given a tree (v_1, v_2), we must first perform some computation on v_1. Namely, it is enough to know, after this computation, if v_1 is in X_1 or not, and similarly for X_3. It is not necessary to do any other computation; for instance, we don't care whether v_1 is in X_2 or not. According to the presence of v_1 in X_1 and/or X_3, we continue with different computations of v_2:

- It v_1 is neither in X_1 nor in X_3, we already know that v is not in X without looking at v_2. We can stop the computation immediately.
- If v_1 is in X_1 but not in X_3, we have to check whether v_2 is in X_2.
- If v_1 is in X_3 but not in X_1, we have to check whether v_2 is in X_4.
- If v_1 is in X_1 and in X_3, we must check whether v_2 is in X_2 or not, and in X_4 or not. But actually, this is too much work, we only have to find whether it is in $X_2 \cup X_4$ or not, and this can be easier to do (for instance, if $X_2 \cup X_4 = \mathscr{V}$, we don't have anything to do at all).

This is the general case, but in some special cases, it is not necessary to know both wgther v_1 is in X_1 *and* whether it is in X_3. For instance, imagine that $X_2 = X_4$. Then we don't have to distinguish the three cases $v_1 \in X_1 \backslash X_3$, $v_1 \in X_3 \backslash X_1$, $v_1 \in X_1 \cap X_3$. Indeed, we only need to check whether v_1 is in $X_1 \cup X_3$ or not. We could as well have merged $X_1 \times X_2$ and $X_3 \times X_4$ into $(X_1 \cup X_3) \times X_2$ in this case. We can also merge them if one is a subset of the other.

Now imagine we have some static information X_0. If for instance, $X_0 \cap (X_1 \times X_2) = \varnothing$, we can simply ignore the rectangle $X_1 \times X_2$. Also, in general, we deduce some information about v_1: it belongs to $\pi_1(X_0) = \{v_1^0 \mid (v_1^0, v_2^0) \in X_0\}$. After performing some computation on v_1, we get more information. For instance, we may deduce $v_1 \in X_1 \backslash X_3$. Then we know that v_2 is in $\pi_2(X_0 \cap (X_1 \backslash X_3) \times \mathscr{V})$. In general, we can combine the static information and the results we get for the a left subtree to get a better static information for the right subtree. Propagating a more precise information allows to ignore more rectangles.

The static information allows us to weaken the condition to merge two rectangles $X_1 \times X_2$ and $X_3 \times X_4$. Indeed, it is enough to check whether $\pi_2(X_0 \cap (X_1 \times X_2)) = \pi_2(X_0 \cap (X_3 \times X_4))$ (which is strictly weaker than $X_2 = X_4$).

In some cases, there are decisions to make. Imagine that $X_0 = X_1 \times X_2 \cup X_3 \times X_4$, and we want to check if a tree (v_1, v_2) is in $X_1 \times X_2$. If we suppose that $X_1 \cap X_3 = \varnothing$ and $X_2 \cap X_4 = \varnothing$, we can work on v_1 to see if is in X_1 or not, or we can work on v_2 to see if is in X_2 or not. We don't need to do both, and we must thus choose which one to do. We always choose to perform some computation on v_1 if it allows to gain useful knowledge on v. This choice allows to stop the top-down left-to-right traversal of the tree as soon as possible. This choice is relevant when considering the encoding of XML sequences and trees in our binary trees. Indeed, the choice correspond to: (1) extracting information from an XML tag to guide the computation on the content of the element, and (2) extracting information from the first children before considering the following ones.

3.2 Types

We have several regular languages $X_0, X_1, \ldots, X_n$ as inputs, and our algorithm needs to produce other languages as intermediate steps.

Instead of working with several different NDTA to define these languages, we assume that all the regular languages we will consider are defined by the same fixed NDTA $\mathscr{A}$ (each language is defined by a specific state of this NDTA). This assumption is not restrictive since it is always possible to take the (disjoint) union of several NDTA. Moreover, we assume that this NDTA has the following properties:

- **Boolean-completeness.** The class of languages defined by $\mathscr{A}$ (that is, the languages of the form $\mathscr{A}[\![r]\!]$), is closed under boolean operations (union, intersection, complement with respect to $\mathscr{V}$).
- **Canonicity.** If $(r_1, r_2, r) \in \delta$, then: $\mathscr{A}[\![r_1]\!] \neq \varnothing, \mathscr{A}[\![r_2]\!] \neq \varnothing$. Moreover, if we consider another pair $(r'_1, r'_2) \neq (r_1, r_2)$ such that $(r'_1, r'_2, r) \in \delta$, then $\mathscr{A}[\![r_1]\!] \cap \mathscr{A}[\![r'_1]\!] = \varnothing$ and $\mathscr{A}[\![r_2]\!] \neq \mathscr{A}[\![r'_2]\!]$.

It is well-known that the class of all regular tree languages is closed under boolean operations. The first property says that the class of languages defined by the fixed NDTA $\mathscr{A}$ is closed under these operations. Starting from an arbitrary NDTA, it is possible to extend it to a Boolean-complete one. If r_1, r_2 are two nodes, we write $r_1 \vee r_2$ (resp. $r_1 \wedge r_2$, $\neg r_1$) for some node r such that $\mathscr{A}[\![r]\!] = \mathscr{A}[\![r_1]\!] \cup \mathscr{A}[\![r_2]\!]$ (resp. $\mathscr{A}[\![r_1]\!] \cap \mathscr{A}[\![r_2]\!]$, $\mathscr{V} \backslash \mathscr{A}[\![r_1]\!]$).

The Canonicity property forces a canonical way to decompose the set $\mathscr{A}[\![r]\!]^2$ as a finite union of rectangles of the form $\mathscr{A}[\![r_1]\!] \times \mathscr{A}[\![r_2]\!]$. For instance, it disallows the following situation: $\{(r_1, r_2) \mid (r_1, r_2, r) \in \delta\} = \{(a, c), (b, c)\}$. In that case, the decomposition of $\mathscr{A}[\![r]\!]^2$ given by δ would have two rectangles with the same second component. To eliminate this situation, we can merge the two rectangles, to keep only $(a \vee b, c)$. We also want to avoid more complex situations, for instance where a rectangle in the decomposition of $\mathscr{A}[\![r]\!]^2$ is covered by the union of others rectangles in this decomposition. It is always possible to modify the transition relation δ of a Boolean-complete NDTA to enforce the Canonicity property (first, by splitting the rectangles to enforce non-intersecting first-components, and then by merging rectangles with the same second component).

We will use the word "type" to refer to the nodes of our fixed NDTA $\mathscr{A}$. Indeed, they correspond closely to the types of the CDuce (internal) type algebra, which support boolean operations and a canonical decomposition of products. Note that the set of types is finite, here. We write $[\![t]\!]$ instead of $\mathscr{A}[\![t]\!]$, $\Delta^2(t) = \{(t_1, t_2) \mid (t_1, t_2, t) \in \delta\}$, and $\Delta^0(t) = \{a \mid (a, t) \in \delta\}$. This allows us to reuse the symbols $\mathscr{A}, r, \ldots$ to refer to the NUA we will build, not the NDTA we start from.

3.3 Filters

Even if we start with a single check to perform ("is the tree in X?"), we may have several check to perform in parallel on a subtree ("is v_1 in X_1 and/or in X_3?"); we will call filter a finite set of checks to perform.

A filter is intended to be applied to any tree v from a given language; for such a tree, the filter must compute which of its elements contain v.

Definition 3.1 *Let τ be a type. A τ-filter is a set of types ρ such that $\forall t \in \rho.\ [\![t]\!] \subseteq [\![\tau]\!]$.*
If $\rho' \subseteq \rho$, let $\rho'|\rho$ be a type such that:

$$[\![\rho'|\rho]\!] = [\![\tau]\!] \cap \bigcap_{t\in\rho'} [\![t]\!] \cap \bigcap_{t\in\rho\backslash\rho'} \mathscr{V}\backslash[\![t]\!]$$

(such a type exists thanks to Boolean-completeness.)
The result of a τ-filter ρ for a tree $v \in [\![\tau]\!]$, written v/ρ, is defined by:

$$v/\rho = \{t \in \rho \mid v \in [\![t]\!]\}$$

Equivalently, we can define v/ρ as the only subset $\rho' \subseteq \rho$ such that $v \in [\![\rho'|\rho]\!]$.

Our construction consists in building a NUA whose states are pairs (τ, ρ) of a type τ and a τ-filter ρ. Note that the set of all these pairs is finite, because we are working with a fixed NDTA to define all the types, so there is only a finite number of them.

3.4 Discussion

The type τ represents the static information we have about the tree, and ρ represents the tests we want to perform on a tree v which is known to be in τ. The expected behavior of the automaton is:

$$\forall v \in [\![\tau]\!].\ \mathscr{A}((\tau,\rho), v) = v/\rho$$

Moreover, the state (τ, ρ) can simply reject any tree outside $[\![\tau]\!]$. Actually, we will build a NUA such that:

$$\texttt{Dom}((\tau,\rho)) = [\![\tau]\!]$$

The rest of the section describes how the NUA should behave on a given input. It will thus mix the description of the expected behavior of the NUA at runtime and the (compile-time) construction we deduce from this behavior.

Results In order to have a reduced NUA, we take for the set of results of a given state (τ, ρ) only the $\rho' \subseteq \rho$ that can be actually obtained for an input in τ:

$$R((\tau,\rho)) = \{\rho' \subseteq \rho \mid [\![\rho'|\rho]\!] \neq \varnothing\}$$

Note that ρ' is in this set if and only if there is a $v \in [\![\tau]\!]$ such that $v/\rho = \rho'$.

Left Assume we are given a tree $v = (v_1, v_2)$ which is known to be in a type τ. What can we say about v_1? Trivially, it is in one of the sets $[\![t_1]\!]$ for $(t_1, t_2) \in \Delta^2(\tau)$. We define:

$$\pi_1(\tau) = \bigvee_{(t_1,t_2)\in\Delta^2(\tau)} t_1$$

It is the best information we can find about v_1 [2]. Note that: $[\![\pi_1(\tau)]\!] = \{v_1 \mid (v_1, v_2) \in [\![\tau]\!]\}$.

Now assume we are given a τ-filter ρ that represents the tests we have to perform on v. Which tests do we have to perform on v_1? It is enough to consider those tests given by the $\pi_1(\tau)$-filter:

$$\pi_1(\rho) = \{t_1 \mid (t_1, t_2) \in \Delta^2(t),\ t \in \rho\}$$

This set is indeed a $\pi_1(\tau)$-filter. It corresponds to our choice of performing any computation on v_1 which can potentially simplify the work we have to do later on v_2. Indeed, two different rectangles in $\Delta^2(t)$ for some $t \in \rho$ have different second projections because of the Canonicity property.

This discussion suggests to take:

$$\texttt{left}((\tau,\rho)) = (\pi_1(\tau), \pi_1(\rho))$$

Right Let us continue our discussion with the tree $v = (v_1, v_2)$. The NUA performs some computation on v_1 from the state (τ_1, ρ_1) with $\tau_1 = \pi_1(\tau)$ and $\rho_1 = \pi_1(\rho)$. Let ρ'_1 be the returned result, which is the set of all the types $t_1 \in \rho_1$ such that $v_1 \in [\![t_1]\!]$.

What can be said about v_2? It is in the following type:

$$\pi_2(\tau;\rho'_1) = \bigvee_{(t_1,t_2)\in\Delta^2(\tau)\ \mid\ [\![t_1\wedge(\rho'_1|\rho_1)]\!]\neq\varnothing} t_2$$

This type represents the best information we can get about v_2 knowing that $v \in [\![\tau]\!]$ and $v_1 \in [\![\rho'_1|\rho_1]\!]$. Indeed, its interpretation is:

$$\{v_2 \mid (v_1, v_2) \in [\![\tau]\!], \rho'_1 = v_1/\rho_1\}$$

Now we must compute the checks we have to perform on v_2. Let us consider a given type $t \in \rho$. If $(t_1, t_2) \in \Delta^2(t)$, we have $t_1 \in \rho_1$, so we know if $v_1 \in [\![t_1]\!]$ or

[2]Here we use the assumption that the rectangles in the decomposition are not empty - this is part of the Canonicity property.

not (namely, $v_1 \in [\![t_1]\!] \iff t_1 \in \rho'_1$). There is at most one pair $(t_1, t_2) \in \Delta^2(t)$ such that $v_1 \in [\![t_1]\!]$. Indeed, two rectangles in the decomposition $\Delta^2(t)$ have non-intersecting first projection (Canonicity). If there is such a pair, we must check if v_2 is in $[\![t_2]\!]$ or not, and this will be enough to decide if v is in $[\![t]\!]$ or not. We thus take:

$$\pi_2(\rho; \rho'_1) = \{t_2 \mid (t_1, t_2) \in \Delta^2(t), t \in \rho, t_1 \in \rho'_1\}$$

The cardinal has at most as many elements as ρ by the remark above. Finally, the "right" transition is:

$$\texttt{right}((\tau, \rho), \rho'_1) = (\pi_2(\tau; \rho'_1), \pi_2(\rho; \rho'_1))$$

Computing the result We write $\tau_2 = \pi_2(\tau; \rho'_1)$ and $\rho_2 = \pi_2(\rho; \rho'_1)$. We can run the NUA from this state (τ_2, ρ_2) on the tree v_2, and get a result $\rho'_2 \subseteq \rho_2$ collecting the $t_2 \in \rho_2$ such that $v_2 \in [\![t_2]\!]$. For a type $t \in \rho$, and a rectangle (t_1, t_2) in its decomposition $\Delta^2(t)$, we have:

$$v \in [\![t_1]\!] \times [\![t_2]\!] \iff (t_1 \in \rho'_1) \wedge (t_2 \in \rho'_2)$$

So the result of running the NUA from the state (τ, ρ) on the tree v is:

$$\delta^2((\tau, \rho), \rho'_1, \rho'_2) = \{t \in \rho \mid \Delta^2(t) \cap (\rho'_1 \times \rho'_2) \neq \varnothing\}$$

Result for atomic symbols Finally, we must consider the case when the tree v is a symbol $a \in \Sigma$. The NUA has only to accept for the state (τ, ρ) trees in the set $[\![\tau]\!]$; so if $a \notin \Delta^0(\tau)$, we can let $\delta^0((\tau, \rho), a)$ undefined. Otherwise, we take:

$$\delta^0((\tau, \rho), a) = \{t \in \rho \mid a \in \Delta^0(t)\}$$

3.5 Formal construction

We can summarize the above discussion by an abstract construction of the NUA:

- the set of states are the pairs (τ, ρ) where τ is a type and ρ a τ-filter;
- $R((\tau, \rho)) = \{\rho' \subseteq \rho \mid [\![\rho' | \rho]\!] \neq \varnothing\}$;
- $\texttt{left}((\tau, \rho)) = (\pi_1(\tau), \pi_1(\rho))$ where: $\pi_1(\tau) = \bigvee\{t_1 \mid (t_1, t_2) \in \Delta^2(\tau)\}$ and $\pi_1(\rho) = \{t_1 \mid (t_1, t_2) \in \Delta^2(t),\ t \in \rho\}$;
- $\texttt{right}((\tau, \rho), \rho'_1) = (\pi_2(\tau; \rho'_1), \pi_2(\rho; \rho'_1))$ where: $\pi_2(\tau; \rho'_1) = \bigvee\{t_2 \mid (t_1, t_2) \in \Delta^2(\tau), [\![t_1 \wedge (\rho'_1 | \rho_1)]\!] \neq \varnothing\}$ and $\pi_2(\rho; \rho'_1) = \{t_2 \mid (t_1, t_2) \in \Delta^2(t), t \in \rho, t_1 \in \rho'_1\}$;
- $\delta^2((\tau, \rho), \rho'_1, \rho'_2) = \{t \in \rho \mid \Delta^2(t) \cap (\rho'_1 \times \rho'_2) \neq \varnothing\}$;
- $\delta^0((\tau, \rho), a) = \{t \in \rho \mid a \in \Delta^0(t)\}$ if $a \in \Delta^0(\tau)$ (undefined otherwise)

Once again, it is out of question to actually materialize this NUA. Indeed, we are interested only in the part accessible from a given state (τ, ρ) corresponding to the pattern matching we need to compile. This abstract presentation has the advantage of simplicity (exactly as for the abstract subset construction for the determinization of automata).

Remark 3.2 *This construction has a nice property: the efficiency of the constructed NUA (that is, the positions where it will ignore subtrees of an input) does not depend on the type τ and the types in ρ (which are syntactic objects), but only on the languages denoted by these types. This is because of the Canonicity property. As a consequence, there is no need to "optimize" the types before running the algorithm.*

3.6 Soundness

The following theorem states that the constructed NUA computes what it is supposed to compute.

Theorem 3.3 *The above construction is well defined and explicitly computable. The resulting NUA is* reduced *and it satisfies the following properties for any state (τ, ρ):*

- $\mathit{Dom}((\tau, \rho)) = [\![\tau]\!]$
- $\forall v \in [\![\tau]\!].\ \mathscr{A}((\tau, \rho), v) = v/\rho$

The proof is by induction on trees, and follows the lines of the discussion above.

Remark 3.4 *It is possible to relax the Canonicity property for types and keep a sound compilation algorithm. However, optimality properties (Section 3.9) crucially depends on the simplifications dictated by the Canonicity property.*

3.7 An example

In this section, we give a very simple example of a NUA produced by our algorithm. We assume that Σ contains at least two symbols a,b and possibly others. We consider a type t_a (resp. t_b) which denotes all the trees with only a leaves (resp. b leaves). Our static information τ_0 is $t_a \vee t_b$, and the filter we are interested in is $\rho_0 = \{t_a, t_b\}$. Assuming proper choices for the NDTA that defines the types, the construction gives for the initial state $q_0 = (\tau_0, \rho_0)$:

- $R(q_0) = \{\{t_a\}, \{t_b\}\}$
- $\texttt{left}(q_0) = q_0$
- $\texttt{right}(q_0, \{t_a\}) = (t_a, \{t_a\})$
- $\texttt{right}(q_0, \{t_b\}) = (t_b, \{t_b\})$
- $\delta^2(q_0, \{t_a\}, \{t_a\}) = \{t_a\}$
- $\delta^2(q_0, \{t_b\}, \{t_b\}) = \{t_b\}$
- $\delta^0(q_0, a) = \{t_a\}$
- $\delta^0(q_0, b) = \{t_b\}$
- $\delta^0(q_0, c)$ undefined if $c \neq a, c \neq b$

There is no need to give the transition functions for the states $q_a = (t_a, \{t_a\})$ and $q_b = (t_b, \{t_b\})$ because they each have a single result ($R(q_a) = \{\{t_a\}\}$ and $R(q_b) = \{\{t_b\}\}$), so the NUA will simply skip the corresponding subtrees. Note that the NUA is tail-recursive. Its behavior is simple to understand: it goes directly to the leftmost leaf and returns immediatly. In particular, it traverses a single path from the root to a leaf and ignore the rest of the tree.

As another example, we can consider the functions `f` and g from the introduction, together with the CDuce encoding of XML documents. Our compilation algorithm indeed produces equivalent automata for the two pattern matchings: they directly fetch the root tag and ignore the rest of the tree.

3.8 Implementation

We rely a lot on the possibility of checking emptiness of a type ($[\![t]\!] = \varnothing$). For instance, the definition of $R((\tau, \rho))$ requires to check a lot of types for emptyness. All the techniques developed for the implementation of XDuce and CDuce subtyping algorithms can be used to do it efficiently. In particular, because of caching, the total cost for all the calls to the emptiness checking procedure does not depend on the number of calls (there is a single exponential cost), so they are "cheap" and we can afford a lot of them. CDuce also demonstrates an efficient implementation of the "type algebra" with boolean combinations and canonical decomposition.

The number of states (τ, ρ) is finite, but it is huge. However, our construction proceeds in a top-down way: starting from a given state (τ, ρ), it defines its set of results and its transitions explicitly. Hence we are able to build the NUA "lazily" (either by computing all the reachable states, or by waiting to consume inputs - this is how the CDuce implementation works).

We haven't studied the theoretical complexity of our algorithm, but it is clearly at least as costly as the inclusion problem for regular tree languages. However, in practice, the algorithm works well. It has been successfully used to compile non-trivial CDuce programs.

Preliminary benchmarks [BCF03] suggests very good runtime performances, and we believe that our compilation strategy for pattern matching is the main reason for that.

3.9 Optimality

Remember that one the advantages of NUAs over DTAs is that they can ignore a whole subtree of input when the set $R(q)$ for the current state q is a singleton. We would like to have some *optimality* for the NUA we have built, to be sure that no other construction would yield a more efficient NUA for the same problem. Due to the lack of space, and because this part is work in progress, we keep the presentation informal.

First, we make precise the notion of information. We say that an information is a partial equivalence relation (PER) $\equiv$ on $\mathscr{V}$ (that is, an equivalence relation whose domain $\texttt{Dom}(\equiv)$ is a subset of $\mathscr{V}$). We define an ordering on PERs. Let $\equiv_1$ and $\equiv_2$ two PERs. We say that the information $\equiv_2$ is larger than $\equiv_1$ and we write $\equiv_1 \leq \equiv_2$ if either:

- the domain of $\equiv_2$ is a strict subset of the domain of of $\equiv_1$: $\texttt{Dom}(\equiv_2) \subsetneq \texttt{Dom}(\equiv_1)$).

- or they have the same domain, and $\equiv_2$ is coarser than $\equiv_1$: $v_1 \equiv_1 v_2 \Rightarrow v_1 \equiv_2 v_2$.

Now we define the information of a state q in a NUA $\mathscr{A}$ as the PER $\equiv_q$ with domain $\mathtt{Dom}(q)$ defined by:

$$v_1 \equiv_q v_2 \iff \mathscr{A}(q, v_1) = \mathscr{A}(q, v_2)$$

Let us give the intuition about the ordering on PERs. The idea is that the domain of a PER represents the information we have before doing any computation (static information), and the partition itself represents the information we extract by doing some computation on a tree (namely, finding the class of the PER the tree belongs to). We want to minimize both the static information we propagate and the amount of computation we require, so we want a NUA to traverse states with largest possible PERs in its traversal of a tree[3].

Here we need to take the traversal order into account, because we have made the following choice: when facing a tree $v = (v_1, v_2)$, the NUA we have built in previous sections extracts any information from v_1 that allows it to get more precise static information on v_2 (using the static information it has on v and the result of the computation on v_1).

For an arbitrary NUA $\mathscr{A}$, we have defined the result of $\mathscr{A}$ on an input v from a state q. In this section, we need to consider that running a NUA annotates the tree. For each subtree, we can define a state q such that the NUA entered this subtree in state q. We annotate the subtree with the PER associated to the state q. So we get a tree of PERs. We can flatten it using a right-to-left traversal (opposite to the operation of the NUA), to get a sequence of PERs (whose length correspond to the number of nodes and leaves in the tree v). We call it the trace of $(\mathscr{A}, q)$ on v.

We can compare the runs of two NUAs with initial states $(\mathscr{A}_1, q_1)$ and $(\mathscr{A}_2, q_2)$ (provided that the domains of the initial states contain v). We say that $(\mathscr{A}_1, q_1)$ is better than $(\mathscr{A}_2, q_2)$ for the input v, if the trace of $(\mathscr{A}_1, q_1)$ on v is larger than the trace of $(\mathscr{A}_2, q_2)$, for a lexicographic extension of the ordering on PERs (note that the two traces have the same length).

[3]Note that a state has a trivial PER (a partition with only one class) if and only if it has only one result (provided the NUA is reduced), which is the case that allows the NUAs to stop the traversal.

Now let τ be a type and ρ a τ-filter. Let $\mathscr{A}$ be an arbitrary NUA with an initial state q. We assume that this state extracts enough information from the inputs, as specified by the filter ρ. Formally, we assume the existence of a function $\sigma : R_2(q) \to \mathscr{P}(\rho)$ such that

$$\forall v \in [\![\tau]\!].\ \sigma(\mathscr{A}_2(q, v)) = \{t \in \rho \mid v \in [\![t]\!]\}$$

(or equivalently, that the PER associated to this state q is smaller than the one associated to the state (τ, ρ) in the constructed NUA). We say that $(\mathscr{A}, q)$ is correct for (τ, ρ). The optimality property can now be stated:

Claim 3.5 (Optimality) *Let $\mathscr{A}$ be the NUA built in the previous sections. For any tree $v \in [\![\tau]\!]$, the trace of $(\mathscr{A}, (\tau, \rho))$ on v is better than the trace of any other NUA which is correct for (τ, ρ).*

The proof should follow the lines of the discussion we used to establish the construction. However, we don't have a formal proof of this property yet. The intuition is that the NUA performs as much computation on a left subtree as necessary to get the most precise information on the right subtree (combining static information and the result on the left subtree) - but no more. So it is not possible, under the static knowledge at hand, to extract more static information about the rest of the tree in the traversal of the NUA. Having more information means having less computation to do on the rest of the tree, hence smaller numbers of possible results, and more opportunities for stopping the traversal early.

However, our ordering on PERs makes it a priority to maximize the static information, before minimizing the amount of computation to do. This corresponds to the choice in our algorithm to performs as much computation on a left subtree as necessary to get the most precise information on the right subtree (combining static information and the result on the left subtree). This is motivated by the fact that having more information means having less computation to do on the rest of the tree, hence more opportunities for stopping the traversal early.

But it is not always true that having strictly more information allows us to do strictly less computation, and this depends on the way the atomic cases (dispatch on the value of the leaves) are implemented. Let us give an example [4] . Let $\Sigma = \{a, b, c\}$ and $X_0 = \{a\} \times \Sigma \cup \{b\} \times$

[4]In this example, we manipulate subsets of Σ instead of types for simplicity.

$\{a, b\}$, $X_1 = \{a\} \times \{b, c\} \cup \{b\} \times \{b\}$. Given the static information X_0, we want to recognize X_1. The NUA that we have constructed will start on the left subtree with the filter $\{\{a\}, \{b\}\}$, that is, it wants to know if the left component in a or b (we are necessarily in one of these two cases because of X_0). If it is a, the static information about the right subtree is Σ, and the filter is $\{\{b, c\}\}$. If it is b, the static information about the right subtree is $\{a, b\}$, and the filter is $\{\{b\}\}$. Note that in both cases, it is enough to check if the right subtree is not a, so we're not doing less computation by distinguishing these two cases, even if we have more precise static information for the right subtree. It would be possible to avoid any computation on the left subtree because the information it gives cannot be used to improve the rest of the computation.

Note that this depends on low-level implementation details, namely the way to implement the dispatch for atomic symbols. It could be the case that indeed, the computation in the second case above is more efficient than the one in the first case (because of the representation of transition tables, ...), thus motivating the computation on the left subtree. This kind of situation occurs in the actual CDuce implementation, because of complex basic types (the analog of the symbols in Σ in this presentation): integer intervals, finite or cofinite sets of atoms, ... A more extensive discussion on this issue is left for a future publication.

4 Conclusion

In this paper, we have formalized the core of the compilation algorithm for pattern matching as implemented in CDuce. To simplify the presentation, we have considered only basic trees, and a pure recognition problem (no capture variable).

The actual implementation deals with:

- The full type algebra, including records, infinite basic types, and arrow types.
- The full patern algebra, with capture variables (including non-linear ones), default values, alternation and disjunction patterns, ...

We plan to report on the complete algorithm and study the optimality property in more details in a future publication.

Acknowledgments

I would like to express my best gratitude to Haruo Hosoya for his his help in improving the presentation of this paper.

References

[BCF03] Véronique Benzaken, Giuseppe Castagna, and Alain Frisch. CDuce: an XML-centric general-purpose language. In *ICFP*, 2003.

[FCB02] Alain Frisch, Giuseppe Castagna, and Véronique Benzaken. Semantic subtyping. In *LICS*, 2002.

[GP03] Vladimir Gapeyev and Benjamin Pierce. Regular object types. In *FOOL*, 2003.

[Hos00] Haruo Hosoya. Regular expression types for XML. *Ph.D thesis. The University of Tokyo*, 2000.

[HP02] Haruo Hosoya and Benjamin Pierce. Regular expression pattern matching for XML. *Journal of Functional Programming*, 2002.

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

[Nev02] Frank Neven. Automata theory for XML researchers. In *SIGMOD Record, 31(3), 2002.*, 2002.

[NS98] Andreas Neumann and Helmut Seidl. Locating matches of tree patterns in forests. In *Foundations of Software Technology and Theoretical Computer Science*, pages 134–145, 1998. Extended abstract available at `http://www.informatik.uni-trier.de/~seidl/conferences.html`.