Type-based Optimization for Regular Patterns

Michael Y. Levin Benjamin C. Pierce

Department of Computer and Information Science
University of Pennsylvania
{milevin,bcpierce}@cis.upenn.edu

University of Pennsylvania

Abstract

Pattern matching mechanisms based on regular expressions feature in a number of recent languages for processing XML. The flexibility of these mechanisms demands novel approaches to the familiar problems of pattern-match compilation—how to minimize the number of tests performed during pattern matching while keeping the size of the output code small.

We describe work in progress on a compilation method based on *matching automata*, a form of tree automata specialized for pattern compilation [9], in which we use the *schema* of the value flowing into a pattern matching expression to generate more efficient target code.

1 Introduction

Schema languages such as DTD, XML Schema, and Relax NG have been steadily growing in importance in the XML community. A schema language provides a mechanism for defining the *type* of XML documents; i.e. the set of constraints that specify the structure of XML documents that are acceptable as data for a certain programming task.

Until recently, schema languages have been used mainly for dynamic validation of XML documents. However, a number of recent language designs—many of them descended from the XDuce language of Hosoya, Pierce, and Vouillon [8, 7]—have showed how such schemas can also be used *statically* for type-checking XML processing code. The technical foundation of XDuce is the notion of *regular types*, a mild generalization of nondeterministic top-down tree automata, which correspond to a (generalized) core of most popular schema notations for XML. XDuce also introduced the idea of using *regular patterns*—regular types decorated with variable binders—as a powerful and convenient primitive for dynamic processing of XML structures.

A significant challenge in compiling XDuce and its relatives is understanding how to translate regular pattern matching expressions into a low-level target language efficiently and compactly. One powerful class of techniques that can help achieve this goal relies on using static type information to generate optimized pattern matching code. The work described here aims to integrate type-based optimization techniques with a (high-performance but, so-far, type-insensitive) compilation method for regular patterns based on a flexible tree recognition mechanism called *matching automata* [9]. Matching automata were originally developed for our compiler for Xtatic [4], an extension of C$^\sharp$ with XDuce's regular types and patterns.

After introducing the source and target languages used by our compiler (Section 2), we set the stage for our compilation method by describing a simple (tree-automaton-based) source-to-source optimization used in some versions of XDuce [6].

Our algorithm is obtained (in Section 4) by combining the intuition of this optimization with matching automata. Our algorithm differs from existing type-based approaches in that it does not explore the source patterns from left to right (thus generating code that traverses the input value from left to right). As we show, this strategy may lead to redundant pattern matching code. Our algorithm, on the other hand, employs a heuristic approach that determines which subtree should be examined next in the generated program based

v	::=	`()`	empty sequence	t	::=	`x`	term variable
		`l[v]...l[v]`	non-empty sequence			`()`	empty sequence
						`l[t]`	singleton element
T	::=	`()`	empty sequence type			t_1`,`t_2	concatenation
		`L[T]`	element type			`A(t)`	function call
		T_1`,`T_2	concatenation type			`match x with`	pattern matching
		T_1`|`T_2	union type			`T→t|...|T→t`	
		`T*`	repetition				
		`Any`	wild card	d	::=	`fun A(`T_1 `x) : `T_2 `= t`	function

Figure 1: Source language

on semantic properties of the associated patterns. This allows us to generate more efficient code for many programs.

Section 5 discusses related work, in particular the *non-uniform automata* [3] used in Frisch's implementation of CDuce [1]. Section 6 describes our plans for further research.

2 Background

We begin by briefly sketching the source language (XDuce) and the target language used in our compilation algorithms.

Figure 1 shows the syntax of values, types, terms, and function declarations of XDuce. (This is actually pretty much the whole language!)

Values represent fragments of XML documents. A value is a sequence of elements, each element consisting of an XML tag and a sequence of children elements. For example, the XML fragment `<person><name><john/></name><age><two/></age></person>` is written, in XDuce syntax, as `person[name[john[]],age[two[]]]`. (For brevity, in this paper we omit any discussion of attributes.)

Regular types mirror most core features of familiar schema languages, including concatenation and union types, repetition, wild card, and element types. An element type contains a tag specification and a type describing children. The former can be a literal XML tag or the wild card tag `~`.

The type membership relation $v \in t$ is described by the following rules: first, $() \in ()$; second, $l[v] \in L[T]$ if $v \in T$ and either $L{=}l$ or $L = \sim$; third, $v \in T*$ if v can be decomposed into a concatenation of $v_1 \ldots v_n$ with each $v_i \in T$; fourth, $v \in T_1,T_2$ if v is the concatenation of two sequences v_1 and v_2 such that $v_1 \in T_1$ and $v_2 \in T_2$; fifth, $v \in \texttt{Any}$, and finally, $v \in T_1|T_2$ if $v \in T_1$ or $v \in T_2$.

The term language of XDuce has expressions for building values, function calls, and `match` expressions. A `match` term consists of an input expression and a list of clauses, each consisting of a pattern and a corresponding right hand side expression. The input expression evaluates to a value that is then matched against each of the patterns in turn; the first clause with a matching pattern is selected and its right hand side is evaluated. XDuce's type checker ensures that the clauses of a `match` expression are exhaustive; i.e., at least one of the patterns in the list of clauses is guaranteed to match the input value. For brevity, we omit patterns with variable binding and, hence, use types and patterns interchangeably.

Most of the constructs of the target language are identical to those of the source language. The only difference is that pattern matching is realized by a `case` construct that is simpler than the `match` form of the source language. A `case` expression has the form `case x of` $p_1 \rightarrow e_1 \ldots p_n \rightarrow e_n$ `else` e_0, where patterns can only be of two kinds: `()`, matching the empty sequence, and `l[x],y`, matching a sequence starting with an element tagged by `l` and binding the contents of the first element to `x` and the sequence of the remaining elements to `y`. (Note that although we do not have binding in the source language, it plays a crucial role in the target language—it allows us to perform traversal of subtrees in an arbitrary order. More on that in Section 4.)

```
fun f(Any x) : Any =
  match x with
    | Any, a[] → 1
    | Any → 2
```

(a)

```
fun f(Any x) : Any =
  case x of
    | () → 2
    | a[x],y →
      case x of
        | () →
          case y of
            | () → 1
            else f(y)
        else f(y)
    | ~[_],y → f(y)
```

(b)

```
fun f(T x) : Any =
  case x of
    | ~[_],y →
      case y of
        | a[_],_ → 1
        else 2
```

(c)

Figure 2: A source language program (a); an equivalent target program (b); a target program for a restricted input type (c)

```
fun f(T x) : Any =
  match x with
    | a[b[]],a[Any] → 1
    | a[Any] → 2
```

(a)

```
fun f(T x) : Any =
  match x with
    | ~[b[]],~[Any],Any → 1
    | ~[Any] → 2
```

(b)

```
fun f(T x) : Any =
  match x with
    | ~[Any],~[Any],Any → 1
    | ~[Any] → 2
```

(c)

Figure 3: A difficult case for tree automata: source program (a); equivalent program after type propagation optimization (b); equivalent optimal program (c)

Figure 2 illustrates how a source program (a) is compiled into a target program (b). Matching against `Any,a[]` is equivalent to locating the last element of the input sequence and checking that its tag is `a` and its contents is empty. Since the target language only supports left to right traversal of sequences, the only way to implement this behavior is by using a recursive function that walks the input sequence from the beginning to the end, checks the tag and the contents of the last element, and returns the appropriate result. (In the actual implementation, such uses of recursion must be converted to more efficient iteration constructs in order to avoid stack overflow problems arising from non-tail recursive implementation of function calls.)

The same example can be used to show the crucial effect of type-based optimization. Suppose that the input type in the source program is changed from `Any` to `T = a[],(a[]|b[])`. If the source program is compiled without regard for this input type, it would result in the same target program. But we can do much better. First, there is no longer any need for the recursive loop, since the input sequence is known to contain exactly two elements, and we can simply skip the first element and examine the second. Furthermore, it is unnecessary to check for emptiness, since the input type prescribes both elements to have empty contents. The optimal (in terms of both size and speed) target program corresponding to the source program with the input type restricted to `T` is shown in Figure 2(c).

3 Type-Based Optimization With Tree Automata

In this section, we explain the basic ideas of type-based optimization in the setting of a source-to-source transformation of regular patterns, viewed as tree automata. In what follows, we speak of patterns and tree automata interchangeably.

Like source language patterns, top-down tree automata (TA) perform the function of recognizing regular tree languages. A TA consists of states and transitions. A transition has a source state, a tag, and a pair

of destination states. A state q accepts a value v = a[v_1],v_2 if there is a transition from q tagged with a or ~ such that its two destination states accept v_1 and v_2 respectively. The key features of top-down tree automata are that the tag is checked first before the destination states are evaluated on the left and right subtrees of the input value (hence automata are top-down) and that the subtrees are evaluated independently of each other. As is well known (and as we will see below), this makes it impossible to recognize some regular languages deterministically.

There is a close resemblance between source patterns and tree automata, and Hosoya showed how the former can be converted into the latter [5].

There may be many equivalent tree automata for a given source pattern—some better than others. Consider, for example, the patterns of the program shown in Figure 2(a). When the input type is restricted to T = a[],(a[]|b[]), this program can be converted into the following program (which is optimal in terms of both size and speed).

```
fun f(T x) : Any =
   match x with
      | ~[Any], a[Any], Any → 1
      | Any → 2
```

The pattern of the first clause skips the first element of the input sequence and checks whether the second element is tagged with a; if so, it succeeds. Notice that having the third Any in this pattern is cheaper than not having anything; in the latter case, the TA would have to check that the remainder of the sequence is empty while in the former case no tests are required.

The idea of the optimization algorithm is to simplify a pattern by traversing its structure while keeping track of what type of values can flow into the currently explored subpattern. If we ever find that the input type to a given sub-pattern is a subtype of the sub-pattern itself, then the sub-pattern is guaranteed to succeed at run time, and, so, it can be replaced by a wild card.

For example, consider the pattern p = a[T*],a[S*], and suppose that it is to be matched against values of type a[T?],~[S] for some types T and S. Traversing p from left to right, we first encounter the sub-pattern a[T*]. The input type specifies that only a-labeled elements can be matched against it; hence, the tag can be replaced by ~. Similarly, note that the contents of the first element must be of type T?—a subtype of T*. As a result, T* in the pattern can be replaced by Any. Continuing this type propagation process, we can obtain the following simplified pattern: ~[Any],a[Any],Any.

However, this algorithm often results in suboptimal patterns. Consider, for instance, the program shown in Figure 3(a). Assuming the input type T = (a[b[]],a[Any]) | a[Any], the type propagation algorithm simplifies this program into the program shown in Figure 3(b). Its first pattern is not optimal; it tries to learn unnecessary information about the contents of the first element of the input sequence. (An optimal solution is shown in Figure 3(c); our algorithm in the following section works optimally for this example.) Moreover, some patterns cannot be represented at all by deterministic top-down tree automata. For example, a[b[]],c[] | a[c[]],b[] cannot be recognized by a deterministic top-down TA because it would have no way of distinguishing acceptable values a[b[]],c[] and a[c[]],b[] from unacceptable a[b[]],b[] and a[c[]],c[]. Such patterns *can* be represented deterministically, however, if we enrich tree automata with some additional structure. One such generalization is matching automata, to which we now turn.

4 Type-Based Optimization With Matching Automata

Tree-automaton-based optimization approaches suffer from two inherent limitations. First, with tree automata, it is not possible to process the left and right subtrees sequentially, using information obtained while traversing the former to optimize processing of the latter. Second, tree automata can only return boolean results (accept or reject); whereas, for efficient implementation of pattern matching, we are interested in a mechanism that can signal *which* of the patterns matched the input; this allows us to match several patterns "in parallel" until we gather enough information to distinguish among them. Matching automata were designed precisely to overcome these limitations.

Two features distinguish matching automata from tree automata. First, matching automata incorporate the notion of results; instead of just succeeding or failing, a MA returns an element from a set of possible

```
fun f(T x) : Any =
  match x with
    | a[b[]],c[] → 1
    | a[c[]],b[] → 2
    | Any → 0
```

(a)

```
fun f(T x) : Any =
  case x of
    | a[x],y →
      case x of
        | b[_],_
          case y of
            | c[_],_ → 1
            else 0
        | c[_],_
          case y of
            | b[_],_ → 2
            else 0
        else 0
    else 0
```

(b)

Figure 4: A source program (a); target program corresponding to an equivalent matching automaton (b)

results. Second, while retaining the basic top-down processing style of tree automata, matching automata support an arbitrary—parallel or sequential—order of subtree traversal.

The function of a MA is to process input trees and return results indicating various outcomes of pattern matching. To allow arbitrary traversal of subtrees, matching automata employ transitions with variables. Such transitions are annotated not only with a tag (which must match a label in the input tree) but also with a pair of variables that are bound to the subtrees of the input. For instance, when a transition annotated with `a[x],y` is taken, `x` and `y` get bound to the contents of the first element of the input sequence and to the rest of the sequence respectively. These variables can then be used in other transitions to specify the subtree received by each successor state.

Rather than formally define matching automata here, we use the syntax of target language programs to illustrate our examples. For instance, Figure 4 shows how a source program (a) with input type $\mathtt{T} = \mathtt{\sim[T_2],T_2}$ where $\mathtt{T_2} = \mathtt{\sim[]}$ can be implemented by a deterministic matching automaton (b). The outer `case` expression in the target program corresponds to a matching automaton state with one outgoing transition annotated by `a[x],y` and one successor state that receives the contents of `x`. The successor state is represented by the nested `case` expression and has two outgoing transitions annotated by `b[_],_` and `c[_],_`. They lead to two other states that examine the contents of `y` bound in the first step. Notice how this automaton postpones processing `y` until some information about `x` is learned.

We now sketch a heuristic algorithm that generates efficient matching automata. The algorithm manipulates a data structure that is a generalization of sets of patterns. In it, patterns are arranged into a matrix whose rows and columns are associated with results and variables respectively. More formally, a *configuration* consists of a tuple of distinct variables $\mathtt{<}x_1,\ldots,x_n\mathtt{>}$ and a set of tuples $\{(p_{11}, \ldots, p_{1n}, r_1), \ldots, (p_{m1}, \ldots, p_{mn}, r_m)\}$ each associating a collection of patterns to a result. A configuration can be depicted as follows:

x_1	...	x_n	
p_{11}	...	p_{1n}	r_1
	...		
p_{m1}	...	p_{mn}	r_m

The initial configuration is constructed from one column of patterns computed as a result of intersecting the input type with the patterns of a `match` expression. From this point the input type is not taken into account by the rest of the algorithm; in particular, there is no explicit type propagation along the lines of the TA-based approach. Figures 5(a) and 5(d) are examples of initial configurations for the programs shown in figures 2(a) and 3(a) with the input types $\mathtt{T_1}$ = `a[],(a[]|b[])` and $\mathtt{T_2}$ = `(a[b[]],a[Any]) | a[Any]` respectively.

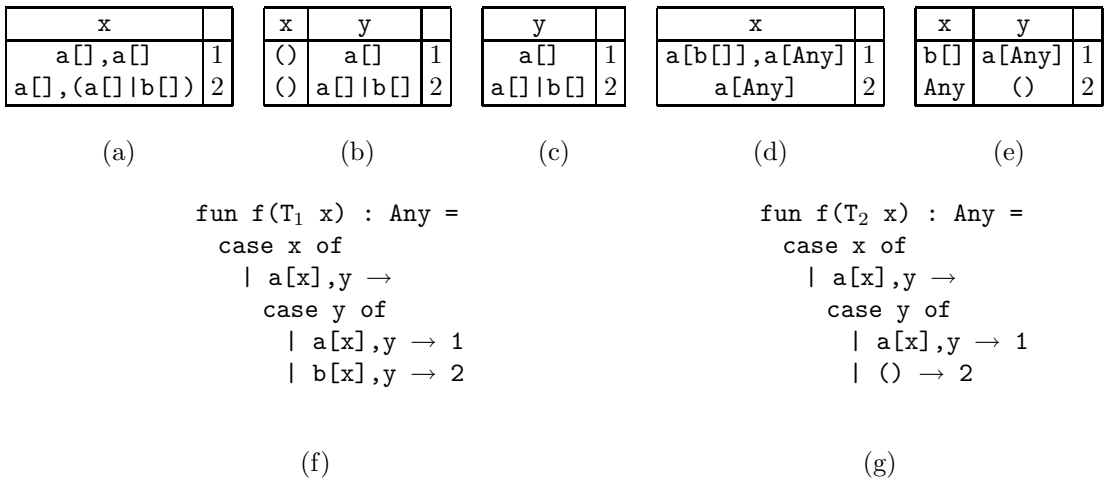

Figure 5: Configurations used in code generation and the obtained target programs

A configuration describes the work that must be done before the outcome of pattern matching can be determined. The variables contain subtrees that have yet to be examined. Pattern matching succeeds with the result stored in some row if all of the row's patterns match the subtrees stored in the corresponding variables.

When faced with a configuration, the compiler has a choice of which subtree (i.e., which column) to examine next. We propose to use the following heuristic. Let P be a set of patterns. A partition of P is *disjoint* if for any two patterns $p_1, p_2 \in P$, if $p_1 \cap p_2 \neq \emptyset$, then both p_1 and p_2 are in the same partition. We say that the *branching factor* of a column is the number of elements in the largest disjoint partition of the column's patterns. The maximal branching factor heuristic then tells us to select the column with the largest branching factor. The motivation behind this heuristic is to arrive at single result configurations as fast as possible. Such configurations need no further pattern matching since the result has already been determined.

There are several simplification techniques for configurations. We have mentioned one already: a single result configuration need not be expanded any further. Another technique involves removing a column all of whose patterns are equivalent. Indeed, because of exhaustiveness, the corresponding subtree necessarily matches all of the column's patterns, and, therefore, no run-time tests are needed.

Figure 5 shows two examples. In the first one, the initial configuration (a) contains patterns matching non-empty a-labeled trees. From this configuration, the compiler generates the pattern `a[x],y` of the outer `case` and proceeds to the next configuration (b). The first column of this configuration is then eliminated because of the simplification technique described above. The resulting configuration (c) is used to generated the clauses of the inner `case`. In the second example, we get to employ the maximal branching heuristic for configuration (e). The branching factor of its first column is one since its patterns are overlapping; the branching factor of the second column, on the other hand, is two. Hence, the inner `case` in Figure 5(g) examines `y` rather than `x`.

Further optimizations are possible. The pattern variables that are not referenced can be replaced by `_`. The last `case` clause can be replaced by a default `else` clause if its pattern does not bind any variables and if there is not already a default clause. A label in a pattern can be replaced by the wild card `~` if the `case` has neither a default clause nor any other label pattern. Applying these simplifications to the first program (f), for example, results in the optimal program shown in Figure 2(c).

Since our heuristic selects the second column of configuration (e), the generated target program is able to skip the subtree stored in `x` completely. This demonstrates an advantage of our approach over the strictly left-to-right type propagation approach of the previous section.

5 Related Work

XDuce [7], a domain-specific language for statically typed XML processing, was the first language to combine regular types and pattern matching. Several successors sprung from XDuce. The goal of one of them, the Xtatic project [4], is to combine the regular types and pattern matching technology with a mainstream object-oriented language. The CDuce language of Benzaken, Castagna, and Frisch [1] is a general purpose language that generalizes XDuce's type system with intersection and function types.

Our previous paper [9] offers a formal treatment of regular pattern compilation. It introduces matching automata and gives a detailed description of two compilation algorithms, constructing backtracking and non-backtracking matching automata. That paper does not address the question of type-based optimization.

Frisch was the first to publish a description of a type-based optimization approach for a XDuce-related language [3]. Frisch's algorithms is based on yet another kind of tree automata—called *non-uniform automata* (NUA). Like matching automata, NUAs incorporate the notion of "results" of pattern matching (i.e., a match yields a value, not just success or failure). Also, like matching automata, NUAs support sequential traversal of subtrees. This makes it possible to construct a deterministic NUA for any regular language. Unlike matching automata, NUAs impose a left to right traversal of the input value. Whereas it is possible for a matching automaton to scan a fragment of the left subtree, continue on with a fragment of the right, come back to the left and so on, an NUA must traverse the left subtree fully before moving on to the right subtree.

Frisch proposes an algorithm that uses type propagation similarly to the tree-automaton-based optimization algorithm in Section 3 above. His algorithm differs from the TA simplification algorithm in that it must traverse several patterns simultaneously (whereas the latter handles one pattern at a time) and generate result sets that will be used in the transitions of the constructed NUA. Similarly to the tree-automaton-based algorithm, Frisch's algorithm does not always achieve optimality. In particular, it generates an NUA that tries to learn as much information from the left subtree as possible, even if this information will not be needed in further pattern matching.

Frisch defines an optimality criterion based on how much information is learned about the input value, and the proposed algorithm is claimed to be optimal in that sense. We prefer an optimality criterion based on the number of tests since it is a more accurate reflection of the running time of the program.

We conjecture that our matching-automaton-based compilation algorithm is a generalization of the NUA-based approach in the sense that the former simulates the latter if, instead of the maximal branching factor heuristic, we use a heuristic that always selects the leftmost column of the current configuration.

Outside of the XDuce family, Fokoue [2] describes a type-based optimization technique for XPath queries. The idea is to evaluate a given query on the schema of the input value obtaining as a result some valuable information that can be used to simplify the query. This is reminiscent of type propagation, but we hesitate to draw deeper analogies since the nature of XPath pattern matching is quite different from that of regular pattern matching.

6 Future Work

We have described a type-based pattern matching compilation algorithm presented in the framework of matching automata. Our technique is designed for compilation into a low-level target language, in contrast to previously existing type-based optimization approaches, which employ automata models that are less suitable for low-level code generation.

This research is a work in progress. In the future, we plan to implement the ideas of this paper in the Xtatic compiler and perform a series of measurements that evaluate the effectiveness of the type-based approach and ascertain the impact of various heuristics used in the algorithm.

We plan to investigate the problem of generating *really* optimal matching automata (both in terms of space and speed). Are these problems (ever) tractable? For example, there exist exponential NFA minimization algorithms [10]. No one (to our knowledge) has investigated how they can be adapted for top-down tree automata or matching automata, and, whether such algorithms can be usable in practice. We leave all these questions for future work.

There are many similarities between our compilation algorithm and Frisch's algorithm for generating efficient non-uniform automata. We would like to study the relationship between these approaches formally.

Does our heuristic algorithm generate strictly more efficient automata than Frisch's type propagation algorithm? Is an optimal matching automaton "more optimal" than an optimal non-uniform automaton for the same matching problem?

Finally, we would like to explore notions of optimality that take into account not only the number of tests but also the number of other low-level operations such as subroutine calls. This will allow us to reason about the quality of the generated matching automaton more realistically.

References

[1] V. Benzaken, G. Castagna, and A. Frisch. CDuce: An XML-centric general-purpose language. In *International Conference on Functional Programming (ICFP), Uppsala, Sweden*, pages 51–63, 2003.

[2] A. Fokoue. Improving the performance of XPath query engines on large collections of XML data, 2002.

[3] A. Frisch. Regular tree language recognition with static information. In *Workshop on Programming Language Technologies for XML (PLAN-X)*, Jan. 2004.

[4] V. Gapeyev and B. C. Pierce. Regular object types. In *European Conference on Object-Oriented Programming (ECOOP)*, 2003. A preliminary version was presented at FOOL '03.

[5] H. Hosoya. *Regular Expression Types for XML*. PhD thesis, The University of Tokyo, Japan, 2000.

[6] H. Hosoya. Implementation of XDuce. `http://xduce.sourceforge.net`, 2004.

[7] H. Hosoya and B. C. Pierce. XDuce: A statically typed XML processing language. *ACM Transactions on Internet Technology*, 3(2):117–148, May 2003.

[8] H. Hosoya, J. Vouillon, and B. C. Pierce. Regular expression types for XML. In *Proceedings of the International Conference on Functional Programming (ICFP)*, 2000.

[9] M. Y. Levin. Compiling regular patterns. In *International Conference on Functional Programming (ICFP), Uppsala, Sweden*, 2003.

[10] O. Matz and A. Potthoff. Computing small nondeterministic finite automata. In E. Brinksma, R. Cleaveland, K. Larsen, and B. Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS '95)*, volume 1019 of *Lecture Notes in Computer Science*, pages 74–88, Aarhus, Denmark, May 1995. Springer-Verlag.