Type-based Optimization for Regular Patterns

Michael Y. Levin Benjamin C. Pierce

Technical Report MS-CIS-05-13
Department of Computer and Information Science
University of Pennsylvania

June 1, 2005

Abstract

Pattern matching mechanisms based on regular tree 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 matching while keeping the size of the output code small.

We describe semantic compilation methods in which we use the *schema* of the value flowing into a pattern matching expression to generate efficient target code. We start by discussing a pragmatic algorithm used currently in the compiler of XTATIC and report some preliminary performance results. For a more fundamental analysis, we define an optimality criterion of "no useless tests" and show that it is not satisfied by XTATIC's algorithm. We constructively demonstrate that the problem of generating optimal pattern matching code is decidable for finite (non-recursive) patterns.

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.

A number of recent designs descended from the XDUCE language of Hosoya, Pierce, and Vouillon [13, 12] have showed how schemas can be used both *statically* for type-checking XML processing code and *dynamically* for evaluation of XML structures. At the core of these languages is the notion of *regular patterns*, a powerful and convenient mechanism for dynamic inspection of XML values.

A significant challenge in compiling languages with regular patterns 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 the high-performance, but type-insensitive, compilation methods described in our previous paper [14]. The ideas developed in this paper are used in the compiler for XTATIC—an object-oriented language with regular types and regular pattern matching [7].

A simple example shows the benefits of using type information during compilation. Figure 1 shows how a high-level source program (a) can be compiled into a low-level target program (b) without taking the input type into account. The first source pattern, `Any,a[]`, matches sequences composed of an arbitrary prefix matching `Any` followed by an `a`-tagged element with the empty contents matching `a[]`. In a low-level target language, this pattern can be implemented by a recursive function that walks the input sequence from the beginning to the end and checks the tag and the contents of the last element. This is precisely the behavior

```
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 1: A source program (a); an equivalent target program (b); a target program for a restricted input type `T = a[],(a[]|b[])` (c)

of the procedure in Figure 1(b). The second clause of the `case` expression, for example, uses the pattern `a[x],y` to check whether the first element in the input sequence is tagged by `a`; then, it employs two nested `case` expressions to ensure that both the contents of the first element and the rest of the sequences are empty. If either is non-empty, the same procedure is invoked recursively on the rest of the input sequence.

However, suppose we know that the input type to the `match` expression is `T = a[],(a[]|b[])`; i.e., only two-element sequences whose first element is tagged by `a` and has the empty contents, and whose second element is tagged by either `a` or `b` and also has the empty contents can be used as pattern matching input. The program shown is Figure 1(b) works correctly for this input type, 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 whether the contents of the second element is empty, since this is prescribed by the input type. The optimal (in terms of both size and speed) target program corresponding to input type `T` is shown in Figure 1(c).

Our contributions are as follows:

- In Section 3 , we present the efficient type-based compilation algorithm used in our implementation of XTATIC and some preliminary measurements that demonstrate the algorithm's effectiveness (compared with XTATIC's previous, type-insensitive compilation method).
- In Section 4, we introduce and justify an optimality criterion that lets us formally compare the efficiency of pattern matching code in target language programs. In Section 5, we demonstrate that optimal compilation is possible, in principle, for matching problems with non-recursive patterns, by presenting a refinement of XTATIC's algorithm that produces optimal target code for this case. (The refined algorithm is too inefficient for use in a real compiler; finding a lower bound on the complexity of optimal compilation is left as future work.) In Section 6, we generalize this algorithm to the case with recursive patterns and show that optimality is not achievable in the general case.

Section 7 discusses related work, in particular the *non-uniform automata* [4] used in Frisch's implementation of CDUCE [1].

2 Background

This section presents the background necessary for the rest of the paper. It describes values and regular patterns, outlines the source and target languages, and introduces tree automata and matching automata.

2.1 Values

A *value* is either the empty sequence `()` or a non-empty sequence of elements, $\texttt{a}_1\texttt{[v}_1\texttt{]} \ldots \texttt{a}_k\texttt{[v}_k\texttt{]}$, each consisting of a label and a nested child value. Values represent fragments of XML documents. For example, the XML fragment `<person><name><john/> </name><age><two/></age></person>` is encoded by the value `person[name[john[]],age[two[]]]`. (For brevity, in this paper we omit any discussion of attributes or pcdata.)

In the rest of the paper, it will be convenient to view values as binary trees. The empty sequence value `()` corresponds to the empty binary tree ϵ. A non-empty sequence value $\texttt{a[v}_1\texttt{],v}_2$ corresponds to the labeled binary tree $a(t_1, t_2)$ whose root label and left and right subtrees correspond to the label of the first element, the child value of the first element, and the rest of the sequence respectively.

We use environments mapping variables to values. We write $E[v_1/x, v_2/y]$ to denote an environment mapping x to v_1 and y to v_2 and agreeing with E on all other variables and $E\backslash y$ to denote an environment which is undefined on y and otherwise equal to E.

It is possible to determine the outcome of many matching problems without traversing the whole input value. The fewer the number of nodes that must be inspected to arrive at the result, the more efficient the corresponding matching automaton can be. To reason about such concerns more easily, we introduce extended values whose nodes are labeled by $+$ or $-$ to indicate whether they are traversed or skipped.

An *annotated value* can be of the form ϵ_* or $a_*(v_1, v_2)$ where v_1 and v_2 are annotated subvalues, l is an element label, and $* \in \{+, -\}$. We say that a value is annotated *consistently*, if for every node of the form $a_-(v_1, v_2)$, both v_1 and v_2 have all their nodes annotated by $-$. A value is *fully traversed* if all its nodes are annotated by $+$. The *erasure* of an annotated value v written $|v|$ is an ordinary value of the same structure with all the annotations eliminated.

Let v_1 and v_2 be consistently annotated values. We say that v_1 is *less traversed* than v_2, written $v_1 \leq v_2$, if $|v_1| = |v_2|$ and, for any node in v_1 labeled by $+$, the corresponding node in v_2 is also labeled by $+$. We say that v_1 is *strictly less traversed* than v_2, written $v_1 < v_2$, if $v_1 \leq v_2$ and $v_1 \neq v_2$.

An *annotated value environment* is a mapping from variable names to annotated values. An environment is *fully traversed* if its range contains only fully traversed values. The erasure operation on annotated value environments $|E|$ producing an ordinary environment is defined pointwise. The $<$ and $\leq$ relations on annotated values are extended point-wise to annotated environments.

2.2 Regular Patterns

Regular patterns are described by the following grammar:

$$p \quad ::= \quad () \mid a[p] \mid p_1, p_2 \mid p_1|p_2 \mid p* \mid \mathit{Any} \mid X$$

These denote the empty sequence pattern, a labeled element pattern, sequential composition, union, repetition, wild-card, and a pattern variable. Pattern variables are introduced by top-level, mutually recursive declarations of the form $\mathit{def}\ X = p$. Top-level declarations induce a function *def* that maps variables to the associated patterns (e.g. the above declaration implies $\mathit{def}(X) = p$.)

The pattern membership relation $v \in p$ is described by the following rules: first, $() \in ()$; second, $a[v] \in a[p]$ if $v \in p$; third, $v \in p*$ if v can be decomposed into a concatenation of $v_1 \ldots v_n$ with each $v_i \in p$; fourth, $v \in p_1, p_2$ if v is the concatenation of two sequences v_1 and v_2 such that $v_1 \in p_1$ and $v_2 \in p_2$; fifth, $v \in \mathit{Any}$ for any v, and finally, $v \in p_1|p_2$ if $v \in p_1$ or $v \in p_2$.

In this paper, we use the term *regular types* synonymously with *regular patterns*. In a full-blown source language, regular patterns may also contain variables used to extract fragments of the input value. Here, for simplicity, we omit patterns with variable binding and identify regular types and regular patterns.

2.3 Source Language

The source language used in our examples has primitives for building values, function calls, and `match` expressions. A `match` expression 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. The 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.

2.4 Target Language

To illustrate how pattern matching is compiled, we employ a target language all of whose constructs except the pattern matching ones are identical to the corresponding constructs of the source language. Pattern matching is realized by a `case` construct which has the same form as the `match` construct of the source language except that patterns can only be of two kinds: `()`, matching the empty sequence, and `a[x],y`, matching a sequence starting with an element tagged by `a` and binding the contents of the first element to x and the sequence of the remaining elements to y. If the tag of the first element need not be examined during the rest of pattern matching, it can be replaced by `~`; if the contents of a variable need not be examined, it may be replaced by the wild card `_`. (See a target language example in Figure 1.)

2.5 Tree Automata

The semantics of regular patterns does not directly give rise to an efficient pattern matching algorithm. To give us a better starting point for generating efficient pattern matching code, we convert source patterns into a form with a substantially simpler pattern matching semantics. This form of patterns can be described by states of a *non-deterministic top-down tree automaton*.

2.1 Definition: A *non-deterministic top-down tree automaton* is a tuple $A = (S, T)$, where S is a set of states and T is a set of transitions consisting of *empty* transitions of the form $s \to ()$ and *label* transitions of the form $s \to a[s_1], s_2$, where $s, s_1, s_2 \in S$ and a is a label. The acceptance relation on values and states, denoted $v \in s$, is defined by the following rules:

$$\frac{s \to () \in T}{() \in s}\ \text{(TA-Emp)} \qquad \frac{s \to a[s_1], s_2 \in T \qquad v_1 \in s_1 \qquad v_2 \in s_2}{a[v_1], v_2 \in s}\ \text{(TA-Lab)}$$

Hosoya and Pierce [11] give a detailed description of the algorithm converting a collection of source patterns into states of a tree automaton. Essentially, it transforms patterns into a disjunctive normal form by applying associativity of concatenation and distributivity of concatenation with respect to union.

From now on, we will assume that all the regular patterns in the source program have been converted to states of one global tree automaton, and we will use these states in place of the corresponding regular types and patterns.

2.6 Matching Automata

Tree automata are not quite appropriate for representing target language pattern matching code. In particular, there is no means to process subtrees of the same node sequentially, there is no explicit support for cycles arising from recursive patterns, and it is hard to match against multiple patterns simultaneously. For all of these reasons, we have previously introduced *matching automata* as a convenient framework for reasoning about the properties of pattern match translations. The function of a matching automaton is to accept a given annotated value and output an integer result depending on its shape. Target language

pattern matching code fragments are isomorphic to a "normalized" form [14] of matching automata that is the target of the compilation process described below.

2.2 Definition: A *matching automaton* is a tuple (Q, q_s, V, R), where Q is a set of states, q_s is a start state, V is a mapping from states to variables, and R is a set of transitions. There are two kinds of transitions: *simple* and *subroutine*. They have the following structure:

$$\begin{array}{ll} q(x) : p \xrightarrow{I} \{q_1 \ldots q_m\} & \text{(simple)} \\ q(x) : A \xrightarrow{\sigma} \{q_1 \ldots q_m\} & \text{(subroutine)} \end{array}$$

Both types of transitions have a *source state* q—associated with some variable x via the mapping V—and a set of *destination states* $\{q_1 \ldots q_m\}$. A simple transition contains a target language pattern p—which can be of the form $()$ or $a[x], z$—and a set of integer results I. A subroutine transition contains a subroutine automaton name A and a binary relation σ over results.

Let $\mathcal{M}$ be a mapping of names to matching automata, and let $A = (Q, q_s, V, R)$ be a matching automaton. The acceptance relation on annotated environments, states, and results, denoted $E \in q \Rightarrow k$, is defined by the following rules:

$$\frac{q(x) : () \xrightarrow{I} \{q_1 \ldots q_m\} \in R \qquad E(x) = \epsilon_+ \qquad k \in I \qquad E' = E \backslash x \qquad \forall i \in \{1 \ldots m\}.\ E' \in q_i \Rightarrow k}{E \in q \Rightarrow k} \tag{MA-Emp}$$

$$\frac{q(x) : a[y], z \xrightarrow{I} \{q_1 \ldots q_m\} \in R \qquad E(x) = a_+(v_1, v_2) \qquad k \in I \qquad E' = (E \backslash x)[v_1/y, v_2/z] \qquad \forall i \in \{1 \ldots m\}.\ E' \in q_i \Rightarrow k}{E \in q \Rightarrow k} \tag{MA-Lab}$$

$$\frac{q(x) : B \xrightarrow{\sigma} \{q_1 \ldots q_m\} \in R \qquad E \in \mathcal{M}(B) \Rightarrow j \qquad (j, k) \in \sigma \qquad E \text{ is fully traversed} \qquad E' = E \backslash x \qquad \forall i \in \{1 \ldots m\}.\ E' \in q_i \Rightarrow k}{E \in q \Rightarrow k} \tag{MA-Sub}$$

An annotated environment E is accepted by matching automaton A with result k, written $E \in A \Rightarrow k$, if it is accepted by the automaton's start state: $E \in q_s \Rightarrow k$.

A concrete implementation of pattern matching deals with ordinary rather than annotated values. Initially, we intended annotations to be returned as a *result* of a matching automaton run indicating which parts of the unannotated input value were inspected. We discovered, however, that we can develop a simpler formalism with annotations as part of input values. So, when we say that one automaton accepts $a_+(\epsilon_+, \epsilon_+)$ and another $a_+(\epsilon_-, \epsilon_+)$, we mean that they both accept the same sequence `a[]` but the former performs more inspections than the latter.

The rule MA-Emp says that a state q accepts an environment E yielding a result `k` if: 1) q is the source state for a transition of the form $q(x) : () \xrightarrow{I} \{q_1 \ldots q_m\}$; 2) the variable x associated with q contains the empty sequence ϵ_+, and 3) each destination state accepts the environment obtained from E by removing x's binding yielding the same result k.

The rule MA-Lab describes how a state can accept an environment if the associated variable contains a non-empty sequence value. It is similar to MA-Emp except that the environments used with the destination states are extended with bindings of fresh variables y and z to the left and right subtrees of the input.

For an environment to be accepted by a state with the help of a subroutine transition, it has to be fully traversed. The intent of this requirement is that once a subroutine matching automaton is invoked, we do not attempt to track which nodes are touched by the subroutine automaton, and it is assumed that any part

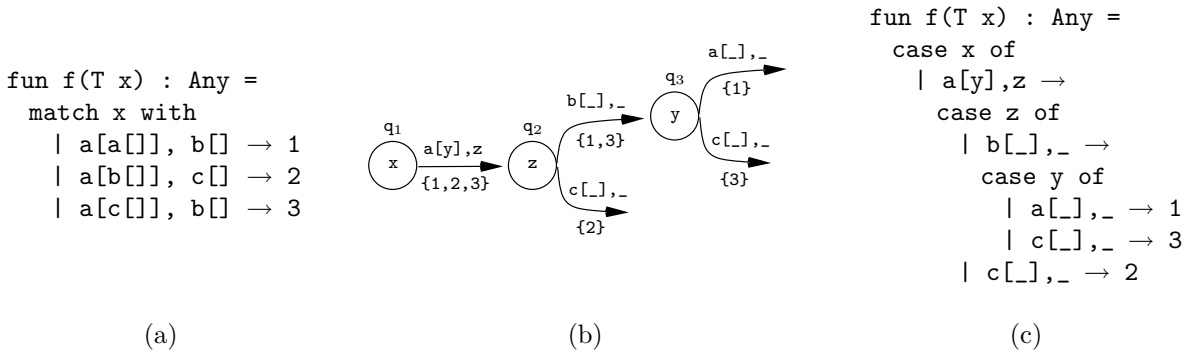


Figure 2: Matching automaton illustration: a source program (a); an equivalent matching automaton (b); an equivalent target program (c); input type `T = a[a[]],b[] | a[b[]],c[] | a[c[]],b[]`

of any input value that has not been processed yet by the current matching automaton may be inspected. According to MA-Sub, an environment is accepted by a state q yielding a result `k` if: 1) there is a subroutine transition of the form $q(x) : B \xrightarrow{\sigma} \{q_1 \dots q_m\}$, the subroutine matching automaton accepts E yielding a result j such that $(j \mapsto k)$ is in the transition's result mapping relation σ, and the destination pairs are checked as in MA-Emp.

The result mapping relations in subroutine transitions serve two purposes. First, they allow us to reduce the number of subroutine automata since we can avoid building isomorphic automata that only differ in their indices. Second, and more importantly, they are essential for creating matching automata that represent non-backtracking target programs.

Figure 2 shows a source program, an equivalent matching automaton, and a corresponding target program. Matching automaton states are depicted with their associated variables inside and their names above the circle. Observe the correspondence between the matching automaton and the target program: states correspond to `case` expressions and transitions to `case` clauses.

2.7 Configurations

During execution of a matching automaton, the current state is faced with an environment mapping variables to values. The following data structure will help us describe the types of values stored in the environment and will be used in the matching automaton generation algorithm.

2.3 Definition: A *configuration* over a tree automaton comprises a tuple of distinct variables $(x_1 \dots x_n)$ and a set of tuples $\{(s_{11} \dots s_{1n}, j_1) \dots (s_{m1} \dots s_{mn}, j_m)\}$, each associating a collection of the tree automaton's states to a result. We depict a configuration by a matrix as follows:

$$C = \begin{array}{|ccc|c|} \hline x_1 & \dots & x_n & \\ \hline s_{11} & \dots & s_{1n} & j_1 \\ & \dots & & \\ s_{m1} & \dots & s_{mn} & j_m \\ \hline \end{array}$$

Two auxiliary functions are defined on configurations: $\mathit{vars}(C) = \{x_1 \dots x_n\}$ and $\mathit{results}(C) = \{j_1 \dots j_m\}$. We say that an ordinary environment E is accepted by C yielding a result j_r, written $E \in C \Rightarrow j_r$, if $E(x_i) \in s_{ri}$ for all $i \in \{1 \dots n\}$. An environment E is accepted by a configuration C, written $E \in C$, if there exists a result j such that $E \in C \Rightarrow j$.

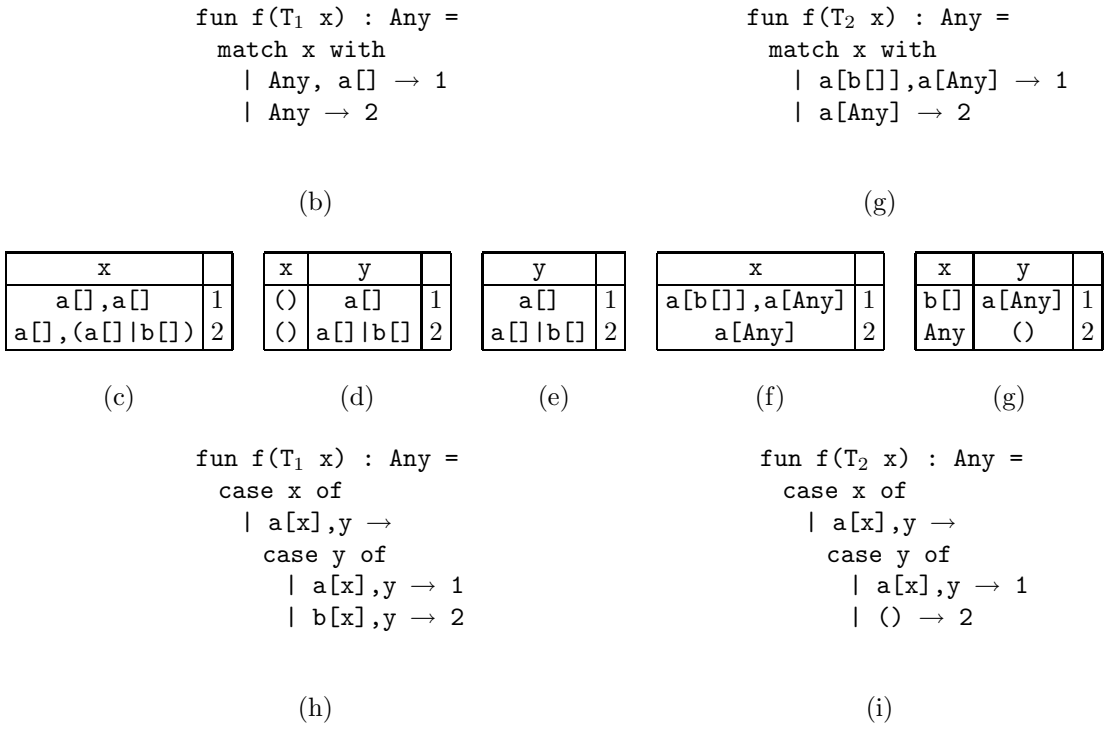


Figure 3: Two source programs (a, b); configurations used in code generation (c - g); and the obtained target programs (h, i). Input types are: $\texttt{T}_1 = \texttt{a[],(a[]|b[])}$ and $\texttt{T}_2 = \texttt{(a[b[]],a[Any]) | a[Any]}$

A configuration describes the work that remains to be done to determine the outcome of pattern matching in a `match` expression. The variables contain subtrees that have yet to be examined. The integer results correspond to the different clauses of the `match` expression.

The notions of boolean operations and subtyping for regular types and tree automaton states can be extend to configurations. Let C and C_0 be a configuration and an input configuration, respectively. For example, we say that $C' = C \cap C_0$ is a configuration such that, for any environment E, if $E \in C_0$ and $E \in C \Rightarrow j$, then $E \in C' \Rightarrow j$.

Unlike the acceptance relation for matching automaton states, the acceptance relation for configurations is defined with respect to ordinary rather than annotated environments. This is because configuration acceptance is expressed in terms of tree automaton state acceptance, and tree automaton states, unlike matching automaton states, traverse input values fully.

3 XTATIC Pattern Compiler

This section presents an efficient type-based compilation algorithm that is used in the current XTATIC compiler. It outlines a general compilation strategy of starting with an initial configuration and gradually expanding it into a completed matching automaton. We describe an effective heuristic for selecting a column on which expansion is based.

The second part of this section summarizes the results of performance experiments for several XTATIC programs. We ran them in the current implementation of XTATIC, recording the size and running time of the generated target programs.

3.1 Heuristic Algorithm

The goal of the compiler is to construct a matching automaton that implements pattern matching in a given `match` expression.

It starts with an *initial configuration* containing the patterns of the `match` expression intersected with the input type. From this point on, the input type is not taken into account. Figures 3(c) and 3(f) are examples of initial configurations for the source programs shown in (a) and (b).

A configuration describes the work that must still be done before the outcome of pattern matching can be determined. The variables contain subtrees that have yet to be examined. Pattern matching will succeed with the result given at the end of 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 use the following heuristic. Let `P` be a set of patterns. A partition of `P` into a number of subsets is *disjoint* if for any two patterns $\mathtt{p}_1, \mathtt{p}_2 \in \mathtt{P}$, if $\mathtt{p}_1 \cap \mathtt{p}_2 \neq \emptyset$, then both $\mathtt{p}_1$ and $\mathtt{p}_2$ are in the same subset. We say that the *branching factor* of a column is the number of subsets 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 involves removing a column all of whose patterns are equivalent. Because of exhaustiveness, the corresponding subtree necessarily matches all of the column's patterns, and, therefore, no run-time tests are needed.

Figure 3 shows two examples. In the first one, the initial configuration (c) 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 (d). The first column of this configuration is then eliminated because of the simplification technique described above. The resulting configuration (e) is used to generated the clauses of the inner `case`. In the second example, we get to employ the maximal branching heuristic for configuration (g). 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 3(i) 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 (h), for example, results in the optimal program shown in Figure 3(a).

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 approaches used in some versions of XDuce and CDuce.

For some examples the heuristic approach falls short of optimality (both informally and in the precise sense defined in Section 4). Consider this configuration:

x	y	z	
a[]	a[]	a[]|b[]	1
a[]|b[]	b[]	a[]|b[]	2
a[]|b[]	a[]|b[]	b[]	3

As we will see in Section 5, it is more beneficial to examine the `y` and `z` columns before examining the `x` column. The heuristic method defined above, however, considers all three columns equal, since they all have a branching factor of 1. So, the heuristic algorithm can potentially generate a suboptimal matching automaton.

Our experience shows that the maximal branching factor heuristic results in high-quality target code for most source programs, since intersection of the input type with `match` patterns and the simple configuration

addrbook		cwn		bibtex	
no tb	tb	no tb	tb	no tb	tb
710	569	19200	17600	35300	26800

(a)

	addrbook		cwn		bibtex	
	no tb	tb	no tb	tb	no tb	tb
n = 1	13 ms	12 ms	300 ms	290 ms	3100 ms	900 ms
n = 2	17 ms	16 ms	420 ms	390 ms	4000 ms	1900 ms
n = 3	23 ms	21 ms	660 ms	510 ms	4900 ms	2900 ms
n = 4	31 ms	28 ms	640 ms	590 ms	11500 ms	4000 ms
n = 5	39 ms	35 ms	770 ms	690 ms	27400 ms	20300 ms

(b)

Figure 4: Size (a) and speed (b) of three source programs with and without type-based optimization; n is a size factor w.r.t. the default input size

optimizations described above seem to account for most type-based optimization opportunities.

3.2 Experiments

To give a sense of the impact of type-based optimization, we compare the performance of three XTATIC programs with and without type-based optimization. The first, `addrbook`, is a small 60 line application that filters an address book and converts the result into a phone book format. The default input for this program is a 31Kb file containing 1,000 address records. We iterate the processing part of the program 10 times to obtain stable results. The second program, `cwn`, converts raw XML newsgroup data into a formatted HTML presentation. The source program contains 400 lines of code; the default input is a 7.7Kb file with seven newsgroup articles. This program is also iterated 10 times. The third program, `bibtex`, is a 700 line program that reads a bibtex file formatted as XML, filters and sorts its contents, and outputs the result as an HTML page. The default input for `bibtex` is a 560Kb file with approximately 1,500 bibtex entries. This processing step of this program is run only once.

Note that XTATIC's compiler is quite efficient even when its type-based optimization is turned off. It employs a variety of other optimizations that go a long way toward producing efficient code. In fact, a previous version of XTATIC's compiler that did not have type-based optimization compared favorably with several other XML processing languages [6].

Figure 4 displays our measurements. Table (a) lists sizes of the output programs in terms of the number of nodes in their ASTs. Table (b) contains running times of the programs for the default input as well as duplicated inputs whose sizes are factors 2, 3, 4, and 5 of the default input's size. Both size and running time measurements are listed for the case when the program was compiled without ("no tb") and with ("tb") type-based optimization as described above.

Overall, these examples illustrate a steady benefit of type-based optimization. It gives us a 10% to 25% improvement in the size of the target program and a similar—or in case of `bibtex` even more dramatic—improvement in the running time. Let us take a closer look at these examples individually.

The `addrbook` program demonstrates a modest improvement in size and speed when compiled with type-based optimization. Figure 5(a) shows a slightly simplified version of `addrbook`'s core fragment. The regular types on the left describe the program's input. Each address book entry contains a `person` element with a `Name`, an optional `Tel`, and a list of `Email`s. Function `mkTelbook` inspects each `Person` entry and checks whether it has a `Tel` subelement. Just using simple configuration optimizations, the XTATIC compiler generates a fairly efficient output code for this program(b). For instance, the produced program does not check the outer `person` label for each input record since from the `match` patterns alone it can be seen that no other label can be expected in this position. The only benefit of type-based analysis here is the ability to infer that the first child of a `person` element must be a `Name`, and, therefore, that there is no need to check for the presence of the `name` label (c). This is precisely what accounts for the better measurements when `addrbook` is compiled with type-based optimization.

In the case of `cwn`, type-based optimization matters less. The only difference of any significance occurs

```
def Name = name[pcdata]
def Tel = tel[pcdata]
def Email = email[pcdata]
def Person = person[Name,Tel?,Email*]
```

```
fun mkTelbook (Person* ps) : Any =
  match ps with
    | person[name[Any],tel[Any],Any], Any → 1
    | person[Any], Any → 2
    | () → 3
```

(a)

```
fun mkTelbook (Person* ps) : Any =
  case ps of
    | () → 3
    | ~[x],y →
      case x of
        | name[z],w →
          case w of
            | tel[u], _ → 1
            | else 2
        else 2
```

(b)

```
fun mkTelbook (Person* ps) : Any =
  case ps of
    | () → 3
    | ~[x],y →
      case x of
        | ~[z],w →
          case w of
            | tel[u],_ → 1
            | else 2
```

(c)

Figure 5: A fragment from `addrbook` example: source program (a); corresponding target language code generated without (b) and with (c) type-based optimization

in a function that performs a character-for-character traversal of its input in order to locate a particular substring. Either a match is found in the beginning of the input or the first character is skipped and the same process is repeated from the next character. Since the input type to this function is `pcdata`—a sequence of character-labeled elements without attributes—there is no need to check for the absence of attributes in every element.

The `bibtex` program gives us the most revealing example of the benefits of type-based optimization. Most of the improvement arises from function `do_xml` that examines the current entry in a bibtex document and determines its type. Figure 6(a) shows the regular types associated with this program. There are fourteen kinds of bibtex entries described by regular type `entry`. The structure of each kind of entry is described by the corresponding regular type (`article` e.g.) Figure 6(b) contains a skeleton of `do_xml`—a dispatch function that branches to different subtasks depending on the kind of the current entry.

Because of the default fall-through case in the `match` expression, a naive compilation strategy that does not take the input type into account results in a *huge* target program that meticulously checks whether the structure of the current element *completely* matches one of the bibtex entry types. In the case of an `article` element, for example, the target program checks whether its contents starts with an `author` element containing `pcdata` and followed by a `bib_title`, `journal`, and `year` elements, and then potentially followed by a `volume` element etc. Using the input type information, however, the compiler realizes that, since only valid `entry` elements can be given as arguments to `do_xml`, and since each entry type has a distinct outer label, checking that outer label is sufficient to determine the type of the entry. Figure 6(c) shows a compact and efficient target program that is the result of compiling `do_xml` with type-based optimization.

4 Optimality Criterion

We have stated in Section 3.1 that XTATIC's heuristic algorithm is not always "optimal". What exactly did we mean by that? This section addresses this by presenting a formal view of optimality. We start by

```
def article =
  article[author, bib_title, journal,
        year, volume?, number?, pages?,
        month?, note?, fields]

def author = author[pcdata]
def bib_title = bib_title[pcdata]
```

```
def entry =
  article | book | booklet | conference |
  inbook | incollection | inproceedings |
  manual | mastersthesis | misc |
  phdthesis | proceedings | techreport |
  unpublished
```

(a)

```
fun do_xml(entry x) : Any =
  match x with
    | article → 1
    | inproceedings → 2
    | unpublished → 3
    | incollection → 4
    | phdthesis → 5
    | techreport → 6
    | book | inbook | manual
    | mastersthesis | proceedings → 7
    | Any → 8
```

(b)

```
fun do_xml(entry x) : Any =
  case x of
    | article[_],_ → 1
    | inproceedings[_],_ → 2
    | unpublished[_],_ → 3
    | incollection[_],_ → 4
    | phdthesis[_],_ → 5
    | techreport[_],_ → 6
    | book[_],_ → 7
    | inbook[_],_ → 7
    | manual[_],_ → 7
    | mastersthesis[_],_ → 7
    | proceedings[_],_ → 7
    else 8
```

(c)

Figure 6: A fragment from `bibtex` example: source program types (a); source language processing function (b); optimal corresponding target program compiled with type-based optimization (c)

observing that "full optimality"—i.e., running at least as fast as any other matching automaton on every input—is not possible. We then define an optimality criterion according to which a program is optimal if there does not exist an equivalent strictly more efficient program. We conclude this section by discussing several limitations of the proposed criterion.

We now turn to a formal discussion of what it means for one target program (or matching automaton) to be better than another one.

Ideally, we would like to perform the minimal number of tests for any input value. Figure 7 demonstrates that this is not always possible. The source program shown in Figure 7(a) contains a `match` expression with three clauses. The clause patterns match sequences starting from `a`-labeled elements. To determine the outcome, the pattern matcher can first investigate the contents of the first element—as in Figure 7(b)—or else look at the rest of the sequence—as in Figure 7(c). In the former case, two tests are required to determine results 1 and 2, but only one test to determine result 3. In the latter case, it takes two tests to determine outcomes 1 and 3 and one to determine result 2. It is not possible for any target language pattern matcher to be as fast as the first program for the input matching the third clause and as fast as the second program for the input matching the second clause.

Consequently, we must settle for near-optimality and, for any pattern matching task, try to build a matcher that is not clearly bested by any other but may not be necessarily *the* best one. First, we recall the formalities of matching automata.

Comparing matching automata is only meaningful with respect to the type of the input values: matching

```
fun f(T x) : Any =
  match x with
    | a[b[]], c[] → 1
    | a[b[]], d[] → 2
    | a[d[]], c[] → 3
```

(a)

```
fun f(T x) : Any =
  case x of
    | ~[y], z →
      case y of
        | b[_], _ →
          case z of
            | c[_], _ → 1
            else 2
        else 3
```

(b)

```
fun f(T x) : Any =
  case x of
    | ~[y], z →
      case z of
        | c[_], _ →
          case y of
            | b[_], _ → 1
            else 3
        else 2
```

(c)

Figure 7: Perfect optimality is unreachable: a source program (a) with input type `T = a[b[]],c[] | a[b[]],d[] | a[d[]],c[]`; a target program that is fast for the third case (b); a target program that is fast for the second case (c)

```
fun f(T x) : Any =
  match x with
    | a[a[]], b[]|c[] → 1
    | a[b[]], c[] → 2
    | a[c[]], b[] → 3
```

(a)

```
fun f(T x) : Any =
  case x of
    | ~[y],z →
      case z of
        | b[_],_ →
          case y of
            | a[_],_ → 1
            | c[_],_ → 3
        | c[_],_ →
          case y of
            | a[_],_ → 1
            | b[_],_ → 2
```

(b)

```
fun f(T x) : Any =
  case x of
    | ~[y],_ →
      case y of
        | a[_],_ → 1
        | b[_],_ → 2
        | c[_],_ → 3
```

(c)

Figure 8: An illustration of optimality criterion: a source program (a) with input type `T = (a[a[]], b[]|c[]) | a[b[]],c[] | a[c[]],b[]`; a suboptimal target program (b); an optimal target program (c)

automaton A may be more efficient than matching automaton B for some set of values but not more efficient—or even not equivalent—for a larger set of values. Intuitively, one matching automaton A is more efficient than another matching automaton B if it needs to traverse a smaller or equal part of an input value to arrive at the same result. This must be the case for any input—A cannot be more efficient than B if it requires more traversal even for one value.

4.1 Definition: Let $M_1 = (Q_1, q_1, V_1, R_1)$ and $M_2 = (Q_2, q_2, V_2, R_2)$ be matching automata, and let C be an input configuration. We say that $q \in Q_1$ is *more efficient* than $q' \in Q_2$ for C, written $C \vdash q \leq q'$, if, for any E such that $|E| \in C$ and $E \in q' \Rightarrow j$, there exists $E' \leq E$ such that $E' \in q \Rightarrow j$. We say that q is *strictly more efficient* than q' for C, written $C \vdash q < q'$ if $C \vdash q \leq q'$ and, for some E such that $|E| \in C$ and $E \in q \Rightarrow j$, it is not the case that $E \in q' \Rightarrow j$. We say that M_1 is more efficient than M_2 for C, written $C \vdash M_1 < M_2$, if $C \vdash q_1 < q_2$.

Consider the example in Figure 8. It shows a source program and two possible translations into the target language. The target program in Figure 8(b) is suboptimal. It tests the right subtree of the input value, and, regardless of the result, inspects the left subtree as well. The program in Figure 8(c) is better—it never inspects the right subtree. This program is more efficient than the suboptimal one since, for any

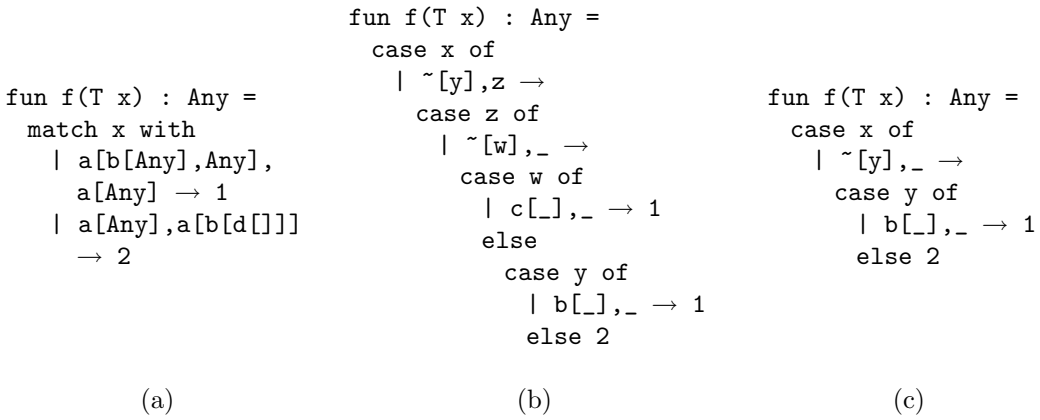


Figure 9: Optimality criterion limitation: a source program (a); with input type `T = (a[b[]], a[b[c[]]]) | (a[Any], a[b[d[]]])`; an optimal target program (b); a *better* optimal target program (c)

annotated value accepted by the latter, the former accepts a less traversed value producing the same result. It is *strictly* more efficient since, for example, $a_+(b_-(\epsilon_-, \epsilon_-), c_+(\epsilon_-, \epsilon_-))$ is accepted by it but not by the suboptimal program.

Note that the proposed measure of optimality does not precisely reflect the amount of work performed by a matching automaton. Consider Figure 9, which shows a source program and two ways of compiling it to the target language. Target program (b) starts by inspecting the right subtree of the input; if it finds a `c` leaf, it can select the first `match` clause; otherwise, it checks whether the root of left subtree is labeled by `b`, and selects the first or the second clause depending on that. Target program (c) checks only the left subtree: if its root is `b`-labeled, it selects the first clause; otherwise the second. The latter program performs fewer than or the same number of node tests as the former for any input. It is *not*, however, any more efficient according to our definition since, for the values matching `a[Any], a[b[c[]]]`, program (b) completely skips the left subtree, while program (c) inspects its root node.

A more precise measure of optimality would involve counting the number of node tests performed by a matching automaton regardless of where in the input value they occur. According to such a measure, program (c) of Figure 9 would be more efficient than program (b). It is difficult, however, to reason about this kind of a measure. For instance, performing various boolean operations such as intersection and difference on regular patterns does not shed any light on how many node tests may be necessary to match a value against them. We leave investigation of this kind of optimality measures for future work.

5 Optimal Compilation for Finite Patterns

Is it possible to generate an optimal matching automaton for a given `match` expression? This section positively answers this questions for a particular class of matching problems—those involving finite (non-recursive) patterns. Building on the intuitions given in Section 3.1—where we informally presented XTATIC's not-always-optimal pattern compiler—we give a formal account of key aspects of the optimal algorithm.

In Section 5.1, we start by introducing an extended version of matching automata—*incomplete matching automata*—in which pairs of configurations can appear as pseudo-states. We then formalize the process of configuration *expansion*, briefly sketched in Section 3.1, and show how using expansion, we can convert a pseudo-state into an ordinary matching automaton state—thus going from the current incomplete matching automaton to a slightly less incomplete matching automaton. Iterating the expansion step will eventually lead to a conventional matching automaton which implements the original pattern matching task.

Section 5.2 pays special attention to a particular way of selecting the expansion column for the current pair of configurations during each iteration of the algorithm. The proposed method is precisely what en-

sures optimality of the generated matching automata. Section 5.3 concludes by establishing the optimality property.

5.1 Incomplete Matching Automata

First, let us introduce an abridged form of configurations without the result column. We will refer to such configurations as *input configurations*. They can be used as a precise specification of the input type for multiple values. Note that an input configuration gives us more information than a simple type assignment. Consider the following configuration:

y	z
T_1	T_2
T_3	T_4

It specifies inputs in which y and z can have respectively either types T_1 and T_2 or types T_3 and T_4. Compare that with the type assignment $\{y \mapsto T_1|T_3, z \mapsto T_2|T_4\}$, which, in addition to the above two scenarios, allows y to be in T_1 while z is in T_4 and y in T_3 while z is in T_2.

Now, we are ready to extend matching automata with pseudo-states. Each pseudo-state is a pair of configurations—one ordinary describing the remaining tests necessary to determine the outcome of pattern matching; the other an input configuration describing the type of the input environment. The initial state can be either a conventional state or a pseudo-state configuration pair.

5.1 Definition: An *incomplete matching automaton* is a tuple (Q, K, i, V, R), where Q, V, and R are a set of states, a mapping from states to variables, and a set of transitions respectively just as in matching automata (Definition 2.2). Additionally, K is a set of pairs of configurations constituting pseudo-states, and i is the initial state which is either an ordinary state $i \in Q$ or a pseudo-state $i \in K$.

The form of transitions is similar to that of matching automaton transitions except that simple transitions can have pseudo-states in addition to states as their destinations. Pseudo-states cannot appear as sources of transitions.

The semantics of simple transitions is defined as in the matching automaton case except that whenever a configuration pair C, D appears in the destination set of a transition, the judgment for configurations $|E| \in C \Rightarrow k$ is used instead of the judgment for states $E \in q \Rightarrow k$:

$$\frac{\begin{array}{c} q(x) : () \xrightarrow{I} \{q_1 \dots q_m, (C_1, D_1) \dots (C_j, D_j)\} \in R \qquad E(x) = \epsilon_+ \qquad k \in I \\ E' = E \backslash x \qquad \forall i \in \{1 \dots m\}.\ E' \in q_i \Rightarrow k \qquad \forall i \in \{1 \dots j\}.\ |E'| \in C_i \Rightarrow k \end{array}}{E \in q \Rightarrow k} \qquad \text{(IMA-Emp)}$$

$$\frac{\begin{array}{c} q(x) : a[y], z \xrightarrow{I} \{q_1 \dots q_m, (C_1, D_1) \dots (C_j, D_j)\} \in R \qquad E(x) = a_+(v_1, v_2) \qquad k \in I \\ E' = (E \backslash x)[v_1/y, v_2/z] \qquad \forall i \in \{1 \dots m\}.\ E' \in q_i \Rightarrow k \qquad \forall i \in \{1 \dots j\}.\ |E'| \in C_i \Rightarrow k \end{array}}{E \in q \Rightarrow k} \qquad \text{(IMA-Lab)}$$

Figure 10 displays several incomplete matching automata in which we omit input configurations from pseudo-states for space reasons. The first example (a) shows an incomplete matching automaton with a single pseudo-state, which is also the initial state. The second (b) is an incomplete matching automaton with an ordinary initial state and a pseudo-state. The automaton contains one transition from the ordinary state to the pseudo-state. We will return to this example below to illustrate the matching automaton generation algorithm.

Now we have all the necessary tools to specify the skeleton of the generation algorithm. The goal is to construct a matching automaton that implements pattern matching in a particular `match` expression. We start with an incomplete matching automaton consisting of one pseudo-state: a pair of an initial configuration

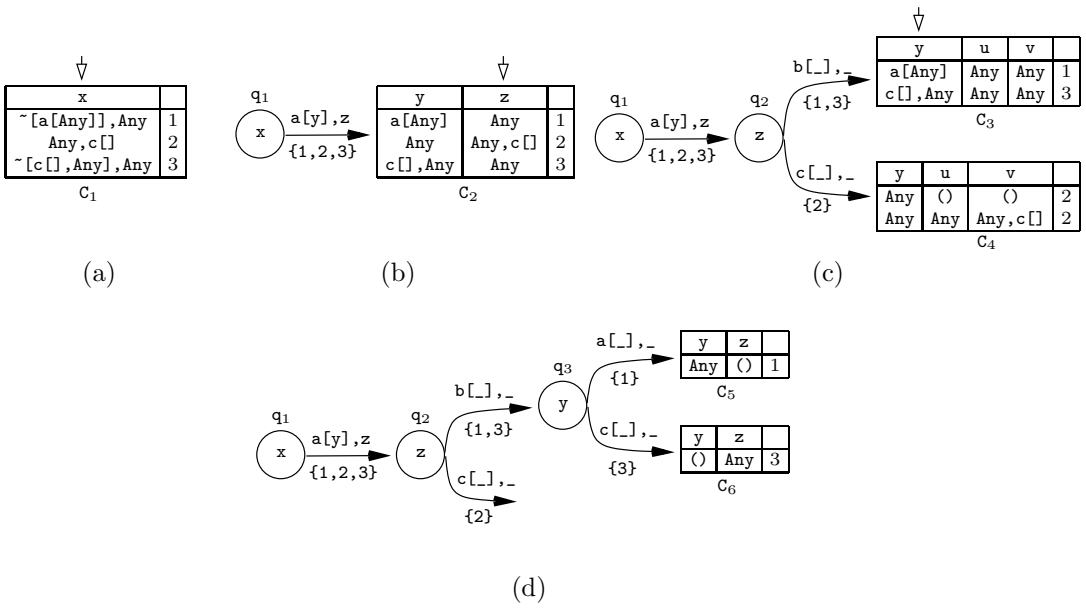


Figure 10: An illustration of a gradual expansion of an input configuration into a matching automaton

C and an initial input configuration D. The former contains a single column composed of the tree automaton states corresponding to the patterns of the `match` expression. The latter contains a single column whose only row has the tree automaton state corresponding to the input type.

Consider the following `match` expression with the input type `T` = `a[a[]],b[] | a[b[]],c[] | a[c[]],b[]` and the corresponding pair of initial configurations:

```
fun f(T x) =
  match x with
    | ~[a[Any]],Any → 1
    | Any,c[] → 2
    | ~[c[],Any],Any → 3
```

x	
~[a[Any]],Any	1
Any,c[]	2
~[c[],Any],Any	3

x
a[a[]],b[] | a[b[]],c[] | a[c[]],b[]

The left configuration describes the pattern matching work that must still be done for the outcome to be determined. The input configuration specifies the type of values stored in the still-to-be-explored variables.

This pair of configurations can be turned into an matching automaton state by the process of *expansion*. For a pseudo-state to become a state, we must determine the variable associated with the new state and all the transitions originating in it. The former is selected among the variables of the configuration and specifies which subtree will be examined next. At this point, we will elide the details of how the selection is made, but we will return to this issue in Section 5.2. The selected variable is indicated by a vertical arrow in the picture. For now, let us concentrate on generating the transitions.

Having selected the column in the current pair of configurations, we can construct all possible target language patterns that match the input type of the selected variable, and, for each pattern, derive a pair of residual configurations that describe the remaining pattern matching work and the types of the still uninspected variables. The example above gives rise to one such target language pattern `a[y],z`, since all the sequences specified by the input type must be non-empty and must start with an `a`-labeled element. The

corresponding residual configuration and input configuration are as follows:

y	z	
a[Any]	Any	1
Any	Any,c[]	2
c[],Any	Any	3

y	z
a[]	b[]
b[]	c[]
c[]	b[]

This expansion step is illustrated in Figure 10(a,b). The obtained incomplete matching automaton (b) is equivalent to the initial incomplete matching automaton (a) and, therefore, correctly implements the original `match` expression.

Proceeding in the same way, we can pick out an unexpanded pseudo-state, expand it, and replace it with the obtained state, transitions, and residual configuration pairs. Eventually, this process will terminate when no more pseudo-states are left. The result will be a complete matching automaton.

Note that threading input configurations throughout the generation process allows us to track what types of values can flow into the currently generated matching automaton state; based on that, we will be able to construct optimal matching automata.

Before we present an outline of the algorithm, we give a formal definition of configuration expansion. The same can be applied for input configurations by dropping result columns.

5.2 Definition: Let $A = (S, T)$ be a tree automaton, and let C be a configuration over A consisting of variables $(x_1 \dots x_n)$ and tuples of tree automaton states $\{(s_{11} \dots s_{1n}, j_1) \dots (s_{m1} \dots s_{mn}, j_m)\}$. Let c be a column in C identified by x_c, and let p be a target language pattern. An *expansion* of C *based on* c *by* p, denoted *expand*(C, c, p), is a configuration C' such that: if $p =$ `()`,

$$C' = \begin{array}{|cccccc|c|} \hline x_1 & \dots & x_{c-1} & x_{c+1} & \dots & x_n & \\ \hline s_{k_1 1} & \dots & s_{k_1(c-1)} & s_{k_1(c+1)} & \dots & s_{k_1 n} & j_{k_1} \\ & & & \dots & & & \\ s_{k_i 1} & \dots & s_{k_i(c-1)} & s_{k_i(c+1)} & \dots & s_{k_i n} & j_{k_i} \\ \hline \end{array}$$

where $\{k_1 \dots k_i\} = \{k \mid s_{kc} \rightarrow \texttt{()} \in T\}$ or, if $p =$ `l[z],y` for some label l and variables $z, y \notin vars(C) \backslash \{x_c\}$,

$$C' = \begin{array}{|cccccccc|c|} \hline z & y & x_1 & \dots & x_{c-1} & x_{c+1} & \dots & x_n & \\ \hline t'_{11} & t''_{11} & s_{11} & \dots & s_{1(c-1)} & s_{1(c+1)} & \dots & s_{1n} & j_1 \\ & & & & \dots & & & & \\ t'_{1k_1} & t''_{1k_1} & s_{11} & \dots & s_{1(c-1)} & s_{1(c+1)} & \dots & s_{1n} & j_1 \\ & & & & \vdots & & & & \\ t'_{m1} & t''_{m1} & s_{m1} & \dots & s_{m(c-1)} & s_{m(c+1)} & \dots & s_{mn} & j_m \\ & & & & \dots & & & & \\ t'_{mk_m} & t''_{mk_m} & s_{m1} & \dots & s_{m(c-1)} & s_{m(c+1)} & \dots & s_{mn} & j_m \\ \hline \end{array}$$

where $\{(t'_{i1}, t''_{i1}) \dots (t'_{ik_i}, t''_{ik_i})\} = \{(t', t'') \mid s_{ic} \rightarrow$ `l[t',t''` $\in T\}$ for $i \in \{1 \dots m\}$.

The following definition formalizes the skeleton of the matching automaton generation algorithm. It is not a complete algorithm, since it does not specify a method for selecting expansion columns in configurations. In the following section, we will discuss how to do optimal column selection. A step in the following algorithm consists of choosing an unexpanded configuration pair, selecting a column in the configurations, expanding the configurations based on the selected column, generating a fresh matching automaton state and a collection of transitions from it to the residual configurations obtained as results of expansion. This step is iterated until there are no more configuration pairs to be expanded; at that point the current incomplete matching automaton is a proper matching automaton.

5.3 Definition: Let $M = (Q, K, i, V, R)$ be an incomplete matching automaton, and let $(C, D) \in K$ be a configuration pair in M where C and D are configurations over tree automaton $A = (S, T)$ sharing the same tuple of variables. C is a ordinary configuration; D is an input configuration. Let c be a column of C identified by x and let d be the corresponding column in D. Let $\{p_1 \ldots p_k\} = \{\texttt{()} \mid s \in d \text{ and } s{\rightarrow}\texttt{()} \in T\} \cup \{\texttt{l[z],y} \mid s \in d \text{ and } \exists\, t', t''.\ s{\rightarrow}\texttt{l[t'],t''} \in T\}$ for some $z, y \notin \mathit{vars}(C)\backslash\{x\}$. Let $C_i = \mathit{expand}(C, c, p_i)$ and $D_i = \mathit{expand}(D, d, p_i)$ and $I_i = \mathit{results}(C_i \cap D_i)$ for each $i \in \{1 \ldots k\}$. Let q be a fresh matching automaton state such that $q \notin Q$.

A *one-step expansion* of M using the configuration pair (C, D) is an incomplete matching automaton $M' = (Q', K', i', V', R')$ where $Q' = Q \cup \{q\}$, and $K' = K\backslash(C, D) \cup \{(C_1, D_1) \ldots (C_k, D_k)\}$, and $i' = i$ if $i \in Q$, or $i' = q$ if $i \in K$, and $V' = V \cup \{q \mapsto x\}$, and $R' = R \cup \{q(x) : p_1 \xrightarrow{I_1} \{(C_1, D_1)\} \ldots q(x) : p_k \xrightarrow{I_k} \{(C_k, D_k)\}\}$. A *complete expansion* of M is a matching automaton obtained by recursively expanding the configuration pairs generated by the previous one-step expansions until there is no more unexpanded configuration pairs.

Returning to the example in Figure 10, we can see the algorithm at work. The end result is the same matching automaton as the one shown in Figure 2. Observe that configurations C_4, C_5, and C_6 are single-result configurations and hence need not be expanded any further. This example also employs the simplification technique of removing a column all of whose patterns are equivalent, as, for instance, the `u` and `v` columns of C_3 do not carry over to the residual configurations C_5 and C_6. Finally, note that we also drop the rows that contain pattern that are incompatible with the input type. This is why C_3 does not have the row with result 2 and C_4 does not have the rows with results 1 and 3.

5.2 Optimal Column Selection

The algorithm we have outlined does not specify how to select expansion columns in configurations for optimal performance. We now complete the algorithm's description by addressing this question. To motivate our column selection approach, we first present a series of examples.

Consider the following configuration with two columns and three results.

y	z	
a[Any]	Any	1
Any	Any,c[]	2
c[],Any	Any	3

Would it be better to test the contents of `y` or `z`? Testing `y` is sufficient to determine the outcome: depending on whether its root node is labeled by a, b, or c, the answer is 1, 2, or 3 respectively. We say that the first column determines all three results. The second column determines only result 2: if the root node of the value stored in `z` is labeled by c, we can conclude 2; if it is labeled by b, however, we cannot determine the result without testing the contents of `y`.

It would seem that testing `y` first would result in more efficient pattern matching, but, in fact, neither column is preferable as far as the optimality measure proposed above is concerned. The reason that expanding on the first column does not lead to a more efficient matching automaton than expanding on the second is that the latter matching automaton can output 2 without considering the contents of `y` at all. We say that neither column is a *better distinguisher* than the other.

If, however, the first row pattern in the second column were changed from `b[]` to `b[]|c[]`, then the second column would not determine any result and, in that case, testing `y` first would be more efficient. The first column in this case would be a better distinguisher than the second column. (Figure 8 shows the two target programs that correspond to choosing `y` or `z` for the initial inspection.)

Sometimes, no single column determines any result. Consider the following configuration.

y	z	
a[]	a[]	1
a[]|b[]	b[]	2
b[]	a[]|b[]	3

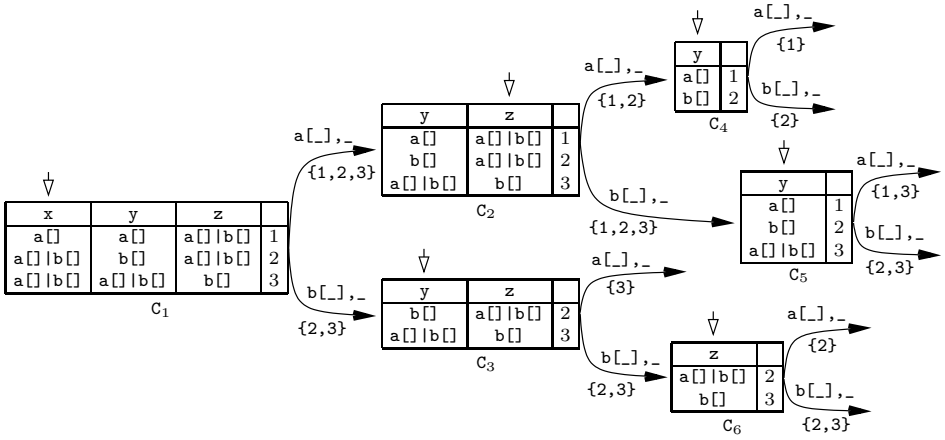


Figure 11: Wrong selection of a column in configuration $\mathtt{C}_1$ leads to a suboptimal matching automaton

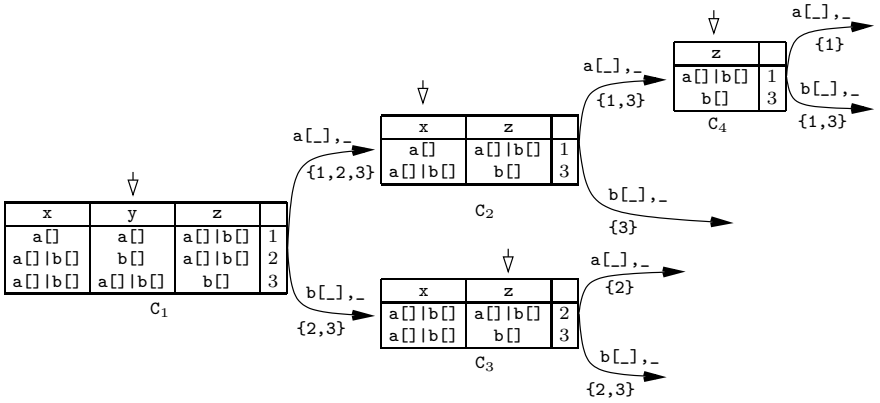


Figure 12: Correct column selection results in a matching automaton that is strictly more efficient than the matching automaton of Figure 11

It is not possible to arrive at the result by testing the contents of either column alone. Of course, testing the contents of both `y` and `z` is sufficient to find the answer. In this case, it does not matter which column is tested first. So, as in the previous example, neither column is the better distinguisher than the other.

The following example shows that even when no column alone determines any result, it is still possible for some column to be better than another. Consider this configuration.

$$C = \begin{array}{|c|c|c|c|} \hline \texttt{x} & \texttt{y} & \texttt{z} & \\ \hline \texttt{a[]} & \texttt{a[]} & \texttt{a[]|b[]} & 1 \\ \texttt{a[]|b[]} & \texttt{b[]} & \texttt{a[]|b[]} & 2 \\ \texttt{a[]|b[]} & \texttt{a[]|b[]} & \texttt{b[]} & 3 \\ \hline \end{array}$$

As in the previous example, testing any of the three columns alone is not sufficient to determine any result. Unlike the previous example, however, it *does* matter which variable we test first. In particular, it can be shown that testing `z` or `y` first is more beneficial than testing `x` first.

Figure 11 shows a potential run of the compilation algorithm for the above configuration. The `x` column is selected for expansion in the first step. When the contents of `x` matches `a[]`, the list of potential pattern matching outcomes cannot be narrowed and further tests must be performed on both `y` and `z`. The matching

automaton generated by the algorithm tests `z` first since C_2 is expanded on the `z` column. Configuration C_3 describes the state of the matching automaton that is reached when `x` matches `b[]`. In this case, potential outcomes are reduced to 2 and 3, and it is possible to conclude 3 by testing whether `y` matches `a[]` and skipping `z` completely. If `y` matches `b[]`, however, `z` must be tested to complete pattern matching.

The matching automaton shown in Figure 11 is suboptimal. This can be seen by comparing it with a more efficient matching automaton shown in Figure 12. In it, testing `y` first allows us not only to conclude 3 without testing `z` as in the previous example but also to arrive at two other outcomes by only testing `y` and `z`—and not `x`—thus outperforming the matching automaton of Figure 11. Similarly, for any other matching automaton that starts by testing `x`, we can always build a strictly more efficient matching automaton that starts by testing one of the other two variables.

For this configuration, we say that both `y` and `z` are *better distinguishers* than `x`. We would like to have a formal criterion that allows us to determine whether one column is a better distinguisher than another. Furthermore, we would like this criterion to be semantic so that we can find an optimally distinguishing column without generating and comparing all possible matching automata that can arise from the current configuration.

We will satisfy the above concerns as follows. First, we will introduce *decision trees*, which have the same semantics as matching automata but are higher level. We will define what it means for one decision tree to be strictly more efficient than another. Then, after establishing a correspondence between decision trees and configurations, we will derive the notion of an optimal expansion column.

5.4 Definition: A *decision tree* is a tree whose nodes are labeled by variables, whose edges are labeled by regular types, and whose leaves are sets of integer results. A path from the root to a leaf may not contain duplicate variables. We say that an environment E is *accepted* by a decision tree t with result j, written $E \in t \Rightarrow j$, if there exists a path $x_1 \xrightarrow{p_1} x_2 \xrightarrow{p_2} \ldots x_k \xrightarrow{p_k} J$ from the root to a leaf, where $x_1 \ldots x_k$ are the variables labeling nodes of the path starting from the root, $p_1 \ldots p_k$ are the regular types labeling the edges of the path, and J is the leaf result set, such that $j \in J$ and $E(x_i) \in p_i$ for all $i \in \{1 \ldots k\}$.

One decision tree is strictly more efficient than another if it accepts any environment by testing a subset of the variables that must be tested by the other decision tree to accept the same environment.

5.5 Definition: A decision tree t_1 is *strictly more efficient* than an equivalent decision tree t_2 if, for any path $x_1 \xrightarrow{p_1} x_2 \xrightarrow{p_2} \ldots x_k \xrightarrow{p_k} J$ in t_2, there exists a path $y_1 \xrightarrow{q_1} y_2 \xrightarrow{q_2} \ldots y_m \xrightarrow{q_m} J$ in t_1 such that, for any $i \in \{1 \ldots m\}$, there exists $j \in \{1 \ldots k\}$ with $y_i = x_j$ and $q_i = p_j$, and, furthermore, there exists a t_2 path for which the corresponding t_1 path is strictly shorter.

A configuration can give rise to a finite number of decision trees. To help identify the set of all decision trees corresponding to a configuration, we first introduce an auxiliary notion of minimal partition of a set of regular types.

5.6 Definition: Let T be an input regular type and $S = \{p_1 \ldots p_m\}$ a set of regular types. A *minimal partition* of S is a minimal set of mutually disjoint regular types $\{t_1 \ldots t_k\}$ such that T is a subtype of $t_1 \cup t_2 \cup \ldots \cup t_k$ and, for any $i \in \{1 \ldots k\}$, there exists $j \in \{1 \ldots m\}$ such that t_i is a subtype of p_j.

A minimal partition of S can be computed by the following algorithm, which repeatedly takes intersections and differences between the types in the candidate result set and the types from S.

```
fun minimal_partition(T,S) =
  // T is the input type
  // S is a set of types to be partitioned
  R = {∅};
  foreach p ∈ S do
    let p = p ∩ T;
    assume R = {t1 ... tk};
    R := {p\(t1 ∪ ... ∪ tk)} ∪ ⋃i∈{1...k}{ti ∩ p, ti\p};
  return R\{∅};
```

Each loop mixes in one additional pattern to the minimal partition of the ones already considered. The intersection identifies the overlap with elements of the previous partition; the difference operations account for the non-overlapping portions of the new pattern and the elements of the previous partition.

Applying this algorithm to any of the three columns of the above configuration C, for instance, will yield the minimal partition $\{\texttt{a[]}, \texttt{b[]}\}$.

5.7 Definition: A decision tree t is said to *correspond* to a configuration C with respect to some input configuration C_0 if two conditions hold: 1) edges from a node x are labeled by regular types each of which is a union of some types from the minimal partition of C's column corresponding to x; and 2) t and C are semantically equivalent; i.e. for any environment $E \in C_0$, we have $E \in t \Rightarrow j$ iff $E \in C \Rightarrow j$.

Given a configuration $C$and an input configuration C_0, it is possible—albeit very time consuming—to generate all decision trees that satisfy the first condition. It is then easy to check whether any such decision tree is semantically equivalent to C. Combining these two steps, we can obtain an algorithm that produces all of C's decision trees.

5.8 Definition: Let C be a configuration, C_0 an input configuration, and c one of C's columns associated with variable x. This column is said to be an *optimal distinguisher* if there exists a decision tree t corresponding to C with respect to C_0 whose root is labeled by x such that there is no other decision tree corresponding to C with respect to C_0 that is strictly more efficient than t.

Figure 13 shows a configuration discussed earlier and two optimal decision trees corresponding to it. The columns associated with z and y are both optimal distinguishers for this configuration. The matching automaton shown in Figure 12 was generated using decision tree (b) as a witness of its optimality.

5.3 Optimality

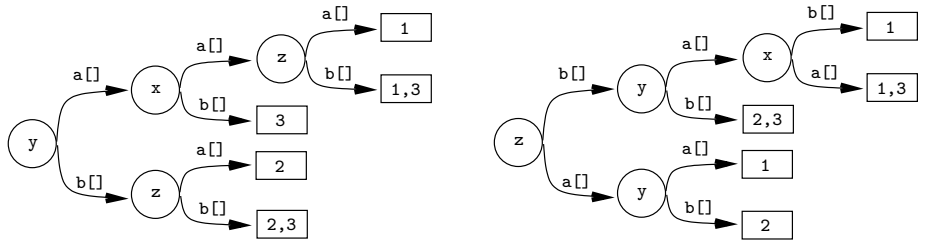
Since the compilation algorithm introduced above can be viewed as an instantiation of the type-insensitive algorithm presented in our previous paper [14]—here the method of selecting expansion columns is specified while there it was left unspecified—the same correctness and termination arguments can be carried over for the algorithm of this paper. Additionally, we can show that the column selection principle introduced above ensures generation of optimal matching automata.

5.9 Lemma: Let C be a configuration and C_0 an input configuration over finite regular types. Let q be the complete expansion of C with respect to C_0, and let M be the associated matching automaton. Then there does not exist a matching automaton with a state that is equivalent to C with respect to C_0 and is strictly more efficient than q.

Proof: Assume that there exists a matching automaton M' with a state q' that is equivalent to C with respect to C_0 and is strictly more efficient than q. If both q and q' use the same test variable, follow the equivalent transitions in both M and M' until a pair of states with different test variables is found. Otherwise, q and q' use different test variables. In any case, let q_1 and q'_1 be the corresponding states with

x	y	z	
a[]	a[]	a[]|b[]	1
a[]|b[]	b[]	a[]|b[]	2
a[]|b[]	a[]|b[]	b[]	3

(a)



(b) (c)

Figure 13: A configuration (a) and two optimal corresponding decision trees (b) and (c)

different test variables where q_1 is in M and q_1' is in M' and q_1' is strictly more efficient than q_1 for q_1's input configuration D_0.

Suppose x and y are q_1's and q_1''s test variables respectively. Let t be the decision tree with x at the root that was used to identify the column associated with x as the expansion column and to generate the transitions originating in q_1 and all the subsequent states of M reachable from q_1. Since q_1' is strictly more efficient than q_1, we can convert the fragment of M' starting at q_1' into a decision tree t' with y at the root that is strictly more efficient than t for input D_0. This is a contradiction, since according to the column selection principle, there cannot be a strictly more efficient equivalent decision tree than t. □

6 Compilation of Recursive Patterns

The algorithm described in the previous section works only for finite patterns; for recursive patterns, it can go into an infinite loop since expansion may not necessarily produce "smaller" residual configurations. Our earlier paper [14] introduced the notion of *loop breakers*—those tree automaton states that may lead to infinite expansion. For configurations with loop breakers, our algorithm used a different expansion technique that produced subroutine transitions and guaranteed termination. The algorithm described in that paper was type-insensitive; as a result, it generated a conservatively large set of loop breakers and less efficient matching automata with many subroutine transitions.

Here, we address several issues of type-based optimization in the presence of recursive patterns. First, we demonstrate that using the input type is beneficial for reducing the number of loop breakers. We then point out that selecting a proper set of loop breakers among a number of candidate sets can make a substantial difference in the efficiency of the resulting matching automaton. We conclude with an observation that it is impossible to achieve optimality as we defined it in the presence of recursive patterns.

6.1 Input Types and Loop Breakers

Computing loop breaker sets without regard for the input type can result in an unnecessarily large number of loop breakers. This can lead to generating superfluous subroutine transitions in cases where simple transitions present a more efficient alternative. Consider, for instance the following configuration that is generated by the `match` expression of the program shown in Figure 1.

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

(a)

```
fun f(T x) : Any =
  case x of
    | ~[y],z →
      case y of
        | ~[_],u →
          case u of
            | a[_],_ →
              let pr = A(z) in
              if pi1(pr) then 1
              else 2
            else 2
```

(b)

```
fun A(Any x) : [bool,bool] =
  case x of
    | () → [false,true]
    | a[y],z →
      case y of
        | () →
          case z of
            | () → [true,true]
            else A(z)
        else A(z)
    | ~[_],z → A(z)
```

(c)

Figure 14: An example with recursive patterns: a source program (a); an equivalent target program with a subroutine call (b); subroutine function (c); input type `T` = `a[(a[]|b[]),(a[]|b[])],Any`

x	
Any,a[]	1
Any	2

The pattern in the first row is recursive; so, if we were to generate a matching automaton that emulates the above configuration for arbitrary input, we would have to treat that pattern as a loop breaker and resort to using subroutine transitions. Knowing that input values values are restricted to the `a[],(a[]|b[])`, however, allows us to avoid recursion. This can be seen if we intersect the input type with the original patterns thus obtaining the following non-recursive configuration.

x	
a[],a[]	1
a[],(a[]|b[])	2

In the type-insensitive algorithm, loop breakers were computed once and for all before matching automaton generation. As we have shown above, such a strategy does not work efficiently in the framework of the type-propagation algorithm described in Section 5 since it leads to unnecessarily identifying many tree automaton states as loop breakers. In the new setting, the loop breaker analysis must be done at each iteration of the algorithm right before the current configuration is expanded. The algorithm computes the intersection of the current configuration and the current input configuration and then finds an appropriate loop breaker set in the obtained configuration.

Consider the example shown in Figure 14. The initial configuration and the initial input configuration corresponding to the `match` expression in the source program are as follows:

$C_1 =$

x	
a[Any,a[]],Any,a[]	1
a[Any],Any	2

$C_2 =$

x
a[(a[]|b[]),(a[]|b[])],Any

Since initial configuration pairs can only be expanded by label, there is no need to perform loop breaker analysis at this point yet. The following configurations C_3 and C_4 are the results of expanding by label the above configurations C_1 and C_2 respectively.

```
def A = a[] | a[B]
def B = b[] | a[A]
def C = c[] | a[C]
def D = ()
def T = a[A],B | a[C]
```

(a)

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

(b)

```
fun f(T x) : Any =
  case x of
    | a[y],z →
      let pr1 = AC(y) in
      let pr2 = BD(z) in
      if pi1(pr1) && pi1(pr2) then 1
      else 2
```

(c)

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

(d)

Figure 15: Effect of selecting loop breakers on optimality: source types (a); a target language processing function (b); an equivalent inefficient target program (c); an optimal target program (d)

$C_3 =$

y	z	
Any,a[]	Any,a[]	1
Any	Any	2

$C_4 =$

y	z
(a[]|b[]),(a[]|b[])	Any

At first glance, both columns of C_3 contain recursive patterns, but if we intersect C_3 with the input configuration C_4, we obtain the following configuration in which only the second column contains loop breakers.

y	z	
(a[]|b[]),a[]	Any,a[]	1
(a[]|b[]),(a[]|b[])	Any	2

The result of expanding this configuration is the program shown in Figure 14(b) in which the contents of `z`—corresponding to the second column—is passed to the subroutine, and the contents of `y`—corresponding to the first column—is inspected inline in the body of `f`.

6.2 Selecting Among Alternative Loop Breaker Sets

Once the current configuration is intersected with the current input configuration and obviously non-recursive patterns are disregarded as potential loop breakers, there still may be multiple ways of choosing a loop breaker set among the rest of the patterns. It is essential for the compiler to choose a loop breaker set that will not lead to an obviously inefficient target program.

As an example, let us evaluate a recursive configuration that arises from the source program shown in Figure 15(b). (Since the input type in this program equals the union of the `match` expression patterns, we will omit discussing input configurations—they are always equivalent to the current configurations.) After expanding the initial configuration, the compiler will encounter the following configuration.

y	z	
A	B	1
C	D	2

```
def T = () | ~[T]
def A = () | a[A]

fun f(T x) : Any =
  match x with
    | A → 1
    | Any → 2
```

(a)

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

(b)

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

(c)

Figure 16: A source program (a); equivalent target program with one recursive call (b); equivalent target program with the recursive call unrolled one level (c)

There are two minimal loop breaker sets: {`A`, `C`} and {`B`, `C`}. If the latter is selected, the above configuration will not have columns without loop breakers and, hence, must be expanded by state. This results in the program shown in Figure 15(c). A much more efficient program can be generated if `A` and `C` are selected. In this case, the above configuration can be expanded by label on its second column. This will produce single result configurations that need not be expanded further hence avoiding subroutine calls altogether (d). Clearly it is beneficial to select loop breakers that reside in the same column of the current configuration, or, in general, minimize the number of the resulting recursive columns.

6.3 Impossibility of Optimality

We conclude this section with the observation that—at least using the current definition of what it means for one matching automaton to be more efficient than another—it is impossible to generate optimal matching automata in the presence of recursive patterns.

Consider the source program in Figure 16. It attempts to identify values of the form `a[a[...a[]...]]` among input values of the form `~[~[...~[]...]]`. The first target program (b) accomplishes this task with the help of a recursive function `f`. The second program (c) is obtained by unrolling the recursive call in the first program once. The two programs perform the same sequence of steps for any input value, and, leaving aside the cost of extra function calls, their performance characteristics are identical. Formally, however, according to Definition 4.1, program (c) is strictly more efficient than program (b). For the empty sequence or any value of the form `a[a[a[...]]]`, both programs traverse the same share of inner nodes. The value matching `a[a[]]` is a different story however. While program (c) accepts annotated value $a_+(a_+(\epsilon_+, \epsilon_-), \epsilon_-)$ skipping the right subtree of the inner a node, program (b) can only accept strictly more traversed $a_+(a_+(\epsilon_+, \epsilon_+), \epsilon_-)$. The reason is that the inner a node is passed as a subroutine argument in program (b) and therefore, according to rule MA-Sub is fully traversed; in program (c), on the other hand, this subroutine call is inlined and, so, the right subtree of the inner a node is manifestly skipped.

Inlining the subroutine call in program (c) will result in an even more efficient program. Continuing this process, we can obtain an infinite sequence of increasingly strictly more efficient programs. This demonstrates that our current formalization of optimality is not appropriate for recursive patterns since it does not guarantee the existence of an optimal matching automaton. To solve this problem, we must change the definition of matching automaton acceptance so that values that are passed to subroutines are not considered fully traversed. We leave exploration of this path to future work.

7 Related Work

Frisch was the first to publish a description of a type-based optimization approach for a language with regular pattern matching [4]. His algorithm is based on a special kind of tree automata called *non-uniform automata*. Like matching automata, non-uniform automata incorporate the notion of "results" of pattern matching (i.e., a match yields a value, not just success or failure). Also, like matching automata, non-uniform automata support sequential traversal of subtrees. This makes it possible to construct a deterministic non-uniform automaton for any regular language. Unlike matching automata, non-uniform automata 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, a non-uniform automaton must traverse the left subtree fully before moving on to the right subtree.

Frisch proposes an algorithm that uses type propagation. His algorithm differs from the tree automaton 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 automaton. Frisch's algorithm does not always achieve optimality. In particular, it generates an automaton 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.

In his dissertation [5], Frisch presents a more flexible form of non-uniform automata that allow arbitrary, rather than strictly left-to-right, order of traversal. There is no formal discussion of optimality however.

Outside of the XDuce family, a lot of work has been done in the area of XPath query optimization. Several subsets of XPath have been considered. Wood describes a polynomial algorithm for finding a unique minimal XPath query that is equivalent to the given query [15]. The minimization problem is solved for the set of all documents regardless of their schema. When the schema is taken into account, the problem is coNP-hard. Flesca, Furfaro, and Masciari consider a wider subset of XPath and show that the minimization problem for it is also coNP-hard [2]. They then identify an subset of their subset for which an ad-hoc polynomial minimization is possible.

Genevès and Vion-Dury describe a logic-based XPath optimization framework [8] in which a collection of rewrite rules is used to transform a query in a subset of XPath into a more efficient, but not necessarily optimal, form.

Optimizing full XPath has also been investigated. Gottlob, Koch, and Pichler observe that many XPath evaluation engines are exponential in the worst case. They propose an algorithm that works for full XPath and that is guaranteed to process queries in polynomial time and space. Furthermore, they define a useful subset of XPath for which processing time and space are reduced to quadratic and linear respectively [9, 10]. Fokoue [3] 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.

At this point, we hesitate to draw deeper analogies between the above XPath-related work and our type-based optimization algorithm since the nature of XPath pattern matching is quite different from that of regular pattern matching. In particular, in XPath models, children of an element are unordered while Xtatic sequences are ordered; also, most subsets of XPath omit disjunctive queries whereas union patterns are prevalent in Xtatic.

References

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

[2] S. Flesca, F. Furfaro, and E. Masciari. On the minimization of xpath queries. In *VLDB*, pages 153–164, 2003.

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

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

[5] A. Frisch. *Théorie, conception et réalisation d'un langage adapté á XML*. PhD thesis, Ecole Normale Supérieure, Paris, Paris, France, 2004.

[6] V. Gapeyev, M. Y. Levin, B. C. Pierce, and A. Schmitt. XML goes native: Run-time representations for Xtatic. In *14th International Conference on Compiler Construction*, Apr. 2005.

[7] V. Gapeyev, M. Y. Levin, B. C. Pierce, and A. Schmitt. The Xtatic experience. In *Workshop on Programming Language Technologies for XML (PLAN-X)*, Jan. 2005. University of Pennsylvania Technical Report MS-CIS-04-24, Oct 2004.

[8] P. Genevès and J.-Y. Vion-Dury. Logic-based XPath optimization. In *International ACM Symposium on Document Engineering*, 2004.

[9] G. Gottlob, C. Koch, and R. Pichler. Efficient algorithms for processing XPath queries. In *VLDB*, pages 95–106, 2002.

[10] G. Gottlob, C. Koch, and R. Pichler. XPath query evaluation: Improving time and space efficiency, 2003.

[11] H. Hosoya and B. C. Pierce. Regular expression pattern matching. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), London, England*, 2001. Full version in *Journal of Functional Programming*, 13(6), Nov. 2003, pp. 961–1004.

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

[13] H. Hosoya, J. Vouillon, and B. C. Pierce. Regular expression types for XML. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(1):46–90, Jan. 2005. Preliminary version in ICFP 2000.

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

[15] P. T. Wood. Minimising simple XPath expressions. In *WebDB*, pages 13–18, 2001.