# XML Goes Native: Run-time Representations for XTATIC

Vladimir Gapeyev, Michael Y. Levin, Benjamin C. Pierce, and Alan Schmitt

University of Pennsylvania

**Abstract.** XTATIC is a lightweight extension of C♯ offering native support for statically typed XML processing. XML trees are built-in values in XTATIC, and static analysis of the trees manipulated by programs is part of the ordinary job of the typechecker. "Tree grep" pattern matching is used to investigate and transform XML trees. XTATIC's surface syntax and type system are tightly integrated with those of C♯. Beneath the hood, however, an implementation of XTATIC must address a number of issues common to any language supporting a declarative style of XML processing (e.g., XQUERY, XSLT, XDUCE, CDUCE, XACT, XEN, etc.). In particular, it must provide representations for XML tags, trees, and textual data that use memory efficiently, support efficient pattern matching, allow maximal sharing of common substructures, and permit separate compilation. We analyze these representation choices in detail and describe the solutions used by the XTATIC compiler.

## 1 Introduction

XTATIC inherits its key features from XDUCE [1, 2], a domain-specific language for statically typed XML processing. These features include XML trees as built-in values, a type system based on *regular types* (closely related to popular schema languages such as DTD and XML-Schema) for static typechecking of computations involving XML, and a powerful form of pattern matching called *regular patterns*. The goals of the XTATIC project are to bring these technologies to a wide audience by integrating them with a mainstream object-oriented language and to demonstrate an implementation with good performance. We use C♯ as the host language, but our results should also be applicable in a Java setting.

At the source level, the integration of XML trees with the object-oriented data model of C♯ is accomplished by two steps. First, the subtype hierarchy of tree types from XDUCE is grafted into the C♯ class hierarchy by making all regular types be subtypes of a special class `Seq`. This allows XML trees to be passed to generic library facilities such as collection classes, stored in fields of objects, etc. Conversely, the roles of tree labels and their types from XDUCE are played by objects and classes in XTATIC; XML trees are represented using objects from a special `Tag` class as labels.

Subtyping in XTATIC subsumes both the object-oriented subclass relation and the richer subtype relation of regular types. XDUCE's simple "semantic" definition of subtyping (sans inference rules) extends naturally to XTATIC's

object-labeled trees and classes. The combined data model and type system, dubbed *regular object types*, have been formalized in [3]. Algorithms for checking subtyping and inferring types for variables bound in patterns can be adapted straightforwardly from those of XDuce ([2] and [4]).

Xtatic's tree construction and pattern matching primitives eschew all forms of destructive update—instead, the language promotes a declarative style of tree processing, in which values and subtrees are extracted from existing trees and used to construct entirely new trees. This style is attractive from many points of view: it is easy to reason about (no need to worry about aliasing), it integrates smoothly with other language features such as threads, and it allows rich forms of subtyping that would be unsound in the presence of update. Many other high-level XML processing languages, including XSLT [5], XQuery [6], CDuce [7], and Xact [8], have made the same choice, for similar reasons. However, the declarative style makes some significant demands on the implementation, since it involves a great deal of replicated substructure that must be shared to achieve acceptable efficiency.

Our implementation is based on a source to source compiler from Xtatic to C♯. One major function of this compiler is to translate the high-level pattern matching statements of Xtatic into low-level C♯ code that is efficient and compact. A previous paper [9] addressed this issue by introducing a formalism of *matching automata* and using it to define both backtracking and non-backtracking compilation algorithms for regular patterns.

The present paper addresses the lower-level issue of how to compile XML values and value-constructing primitives into appropriate run-time representations. We explore several alternative representation choices and analyze them with respect to their support for efficient pattern matching, common Xtatic programming idioms, and safe integration with foreign XML representations such as the standard Document Object Model (DOM). Our contributions may be summarized as follows: (1) a data structure for sequences of XML trees that supports efficient repeated concatenation on both ends of a sequence, equipped with a fast algorithm for calculating the subsequences bound to pattern variables; (2) a compact and efficient hybrid representation of textual data (PCDATA) that supports regular pattern matching over character sequences (i.e., a statically typed form of string grep); (3) a type-tagging scheme allowing fast dynamic revalidation of XML values whose static types have been lost, e.g., by upcasting to `object` for storage in a generic collection; and (4) a proxy scheme allowing foreign XML representations such as DOM to be manipulated by Xtatic programs without first translating them to our representation. (Because of space constraints, we present only the first here; details of the others can be found in an extended version of the paper [10].) We have implemented these designs and measured their performance both against some natural variants and against implementations of other XML processing languages. The results show that a declarative statically typed embedding of XML transformation operations into a stock object-oriented language can be competitive with existing mainstream XML processing frameworks.

The next section briefly reviews the XTATIC language design. The heart of the paper is Section 3, which describes and evaluates our representations for trees. Section 4 summarizes results of benchmarking programs compiled by XTATIC against other XML processing tools. Section 5 discusses related work.

## 2 Language Overview

This section sketches just the aspects of the XTATIC design that directly impact runtime representation issues. More details can be found in [11, 3].

Consider the following document fragment—a sequence of two entries from an address book—given here side-by side in XML and XTATIC concrete syntax.

```
<person>                              [[ <person>
  <name>Haruo Hosoya</name>                 <name>`Haruo Hosoya`</name>
  <email>hahasoya</email>                   <email>`hahasoya`</email>
</person>                                 </person>
<person>                                  <person>
  <name>Jerome Vouillon</name>              <name>`Jerome Vouillon`</name>
  <tel>123</tel>                            <tel>`123`</tel>
</person>                                 </person> ]]
```

XTATIC's syntax for this document is very close to XML, the only differences being the outer double brackets, which segregate the world of XML values and types from the regular syntax of C♯, and backquotes, which distinguish PCDATA (XML textual data) from arbitrary XTATIC expressions yielding XML elements.

One possible type for the above value is a list of persons, each containing a name, an optional phone number, and a list of emails:

```
<person> <name>pcdata</> <tel>pcdata</>? <email>pcdata</>* </person>*
```

The type constructor "`?`" marks optional components, and "`*`" marks repeated sub-sequences. XTATIC also includes the type constructor "`|`" for non-disjoint unions of types. The shorthand `</>` is a closing bracket matching an arbitrarily named opening bracket. Every regular type in XTATIC denotes a set of sequences. Concatenation of sequences (and sequence types) is written either as simple juxtaposition or (for readability) with a comma. The constructors "`*`" and "`?`" bind stronger than "`,`", which is stronger than "`|`". The type "`pcdata`" describes sequences of characters.

Types can be given names that may be mentioned in other types. E.g., our address book could be given the type `APers*` in the presence of definitions

```
regtype Name  [[ <name>pcdata</> ]]
regtype Tel   [[ <tel>pcdata</>  ]]
regtype Email [[ <email>pcdata</> ]]
regtype TPers [[ <person> Name Tel </> ]]
regtype APers [[ <person> Name Tel? Email* </> ]]
```

A *regular pattern* is just a regular type decorated with variable binders. A value `v` can be matched against a pattern `p`, binding variables occurring in `p`

to the corresponding parts of `v`, if `v` belongs to the language denoted by the regular type obtained from `p` by stripping variable binders. For matching against multiple patterns, XTATIC provides a `match` construct that is similar to the `switch` statement of C♯ and the `match` expression of functional languages such as ML. For example, the following program extracts a sequence of type `TPers` from a sequence of type `APers`, removing persons that do not have a phone number and eliding emails.

```
static [[ TPers* ]] addrbook ([[ APers* ]] ps) {
 [[ TPers* ]] res = [[ ]];    bool cont = true;
 while (cont) {
   match (ps) {
    case [[<person> <name>any</> n, <tel>any</> t, any </>, any rest]]:
      res = [[ res, <person> n, t </> ]];   ps = rest;
    case [[ <person> any </person>, any rest ]]:     ps = rest;
    case [[ ]]:    cont = false;  } }
 return res; }
```

The integration of XML sequences with C♯ objects is accomplished in two steps. First, XTATIC introduces a special class named `Seq` that is a supertype of every XML type—i.e., every XML value may be regarded as an object this class. The regular type `[[any]]` is equivalent to the class type `Seq`. Second, XTATIC allows any object—not just an XML tag—to be the label of an element. For instance, we can write `<(1)/>` for the singleton sequence labeled with the integer `1` (the parentheses distinguish an XTATIC expression from an XML tag); similarly, we can recursively define the type `any` as `any = [[ <(object)>any</>* ]]`.

We close this overview by describing how XTATIC views textual data. Formally, the type `pcdata` is defined by associating each character with a singleton class that is a subclass of the C♯ `char` classand taking `pcdata` to be an abbreviation for `<(char)/>*`. In the concrete syntax, we write `‘foo‘` for the sequence type $\texttt{<(char}_f\texttt{)/><(char}_o\texttt{)/><(char}_o\texttt{)/>}$ and for the corresponding sequence value. This treatment of character data has two advantages. First, there is no need to introduce a special concatenation operator for `pcdata`, as the sequence `‘ab‘,‘cd‘` is identical to `‘abcd‘`. This can also be seen at the type level:

```
pcdata,pcdata = <(char)/>*,<(char)/>* = <(char)/>* = pcdata
```

Equating `pcdata` with `string` would not allow such a seamless integration of the string concatenation operator with the sequence operator. Second, singleton character classes can be used in pattern matching to obtain functionality very similar to string regular expressions [12]. For instance, the XTATIC type `‘a‘,pcdata,‘b‘` corresponds to the regular expression `a.*b`.

## 3 Representing Trees

We now turn to the design of efficient representations for XML trees. First, we design a tree representation that supports XTATIC's view of trees as shared and

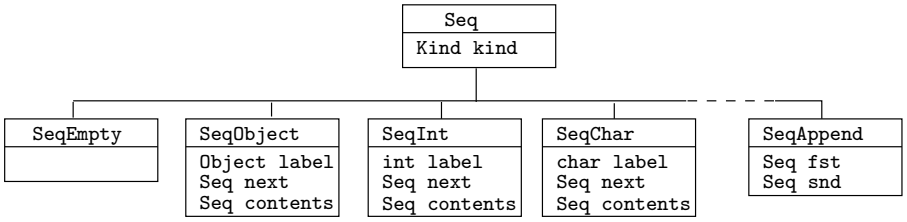


**Fig. 1.** Classes used for representing sequences.

immutable structures (Section 3.1). The main constraint on the design is that the programming style favored by XTATIC involves a great deal of appending (and consing) of sequences. To avoid too much re-copying of sub-sequences, we enhance the naive design to do this appending lazily (Section 3.2). Finally, XTATIC needs to inter-operate with other XML representations available in .NET, in particular DOM. In the full version of this paper [10], we show how DOM structures can masquerade as instances of our XTATIC trees in a type-safe manner.

### 3.1 Simple Sequences

Every XTATIC value with a regular type is a *sequence* of trees. XTATIC's pattern-matching algorithms, based on tree automata, require access to the label of the first tree in the sequence, its children, and its following sibling. This access style is naturally supported by a simple singly linked structure.

Figure 1 summarizes the classes implementing sequences. `Seq` is an abstract superclass representing all sequences regardless of their form. As the exact class of a `Seq` object is often needed by XTATIC-generated code, it is stored as an enumeration value in the field `kind` of every `Seq` object. Maintaining this field allows us to use a `switch` statement instead of a chain of `if-then-else` statements relying on the "`is`" operator to test class membership.

The subclass `SeqObject` includes two fields, `next` and `contents`, that point to the rest of the sequence—the right sibling—and the first child of the node. The field `label` holds a C♯ object. Empty sequences are represented using a single, statically allocated object of class `SeqEmpty`. (Using `null` would require an extra test before `switch`ing on the kind of the sequence—in effect, optimizing the empty-sequence case instead of the more common non-empty case.)

In principle, the classes `SeqEmpty` and `SeqObject` can encode all XTATIC trees. But to avoid downcasting when dealing with labels containing primitive values (most critically, characters), we also include specialized classes `SeqBool`, `SeqInt`, `SeqChar`, etc. for storing values of base types.

XML data is encoded using `SeqObject`s that contain, in their `label` field, instances of the special class `Tag` that represent XML tags. A tag object has a string field for the tag's local name and a field for its namespace URI. We use memoisation (interning) to ensure that there is a single run-time object for each known tag, making tag matching a simple matter of physical object comparison.

```
Seq lazy_norm(Seq node) {
  switch (node.kind) {
    case Append: return norm_rec(node.fst, node.snd);
    default:     return node;   }  }

Seq norm_rec(Seq node, Seq acc) {
  switch (node.kind) {
    case Append: return norm_rec(node.fst, new SeqAppend(node.snd, acc));
    case Object:
      switch node.next.kind {
        case Empty: return new SeqObject(node.label, node.contents, acc);
        default:    return new SeqObject(node.label, node.contents,
                                         new SeqAppend(node.next, acc));  }
    /* similar cases for SeqInt, SeqBool, ... */   }  }
```

**Fig. 2.** Lazy Normalization Algorithm.

Pattern matching of labels is implemented as follows. The object (or value) in a label matches a label pattern when: the pattern is a class `C` and the object belongs to a subclass of `C`, the pattern is a tag and the object is physically equal to the tag, the pattern is a base value `v` and the label holds a value equal to `v`.

### 3.2 Lazy Sequences

In the programming style encouraged by XTATIC, sequence concatenation is a pervasive operation. Unfortunately, the run-time representation outlined so far renders concatenation linear in the size of the first sequence, leading to unacceptable performance when elements are repeatedly appended at the end of a sequence, as in the assignment of `res` in the `addrbook` example in Section 2.

This observation naturally suggests a lazy approach to concatenation: we introduce a new kind of sequence node, `SeqAppend`, that contains two fields, `fst` and `snd`. The concatenation of (non-empty) sequences `Seq1` and `Seq2` is now compiled into the constant time creation of a `SeqAppend` node, with `fst` pointing to `Seq1`, and `snd` to `Seq2`. We preserve the invariant that neither field of a `SeqAppend` node points to the empty sequence.

To support pattern matching, we need a *normalization* operation that exposes at least the first element of a sequence. The simplest approach, *eager* normalization, just transforms the whole sequence so that it does not contain any top-level `SeqAppend` nodes (children of the nodes in the sequence are not normalized). However, there are cases when it is not necessary to normalize the whole sequence, e.g. when a program inspects only the first few elements of a long list. To this end we introduce a *lazy* normalization algorithm, given in pseudocode form in Figure 2.

The algorithm fetches the first concrete element—that is, the leftmost non-`SeqAppend` node of the tree—copies it (so that the contexts that possibly share it are not affected), and makes it the first element of a new sequence consisting of

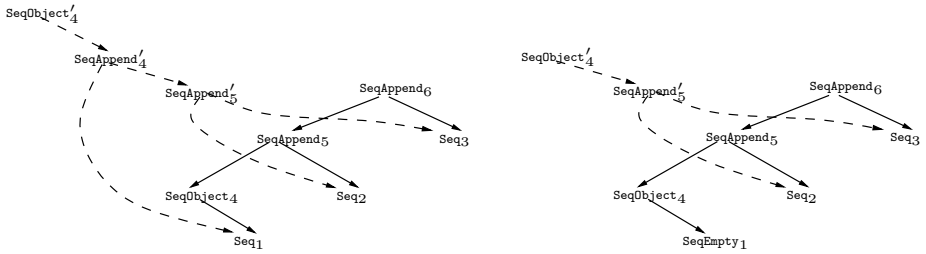


**Fig. 3.** Lazy normalization of lazy sequences. In (a), the leftmost concrete element has a right sibling; in (b) it does not. Dotted pointers are created during normalization.

(copies of) the traversed `SeqAppend` nodes arranged into an equivalent, but right-skewed tree. Figure 3 illustrates this algorithm, normalizing the sequence starting at node $\texttt{SeqAppend}_6$ to the equivalent sequence starting at node $\texttt{SeqObject}'_4$.

Since parts of sequence values are often shared, it is not uncommon to process (and normalize) the same sequence several times. As described so far, the normalization algorithm returns a new sequence, e.g. $\texttt{SeqObject}'_4$, but leaves the original lazy sequence unchanged. To avoid redoing the same work during subsequent normalizations of the same sequence, we also modify *in-place* the root `SeqAppend` node, setting the `snd` field to `null` (indicating that this `SeqAppend` has been normalized), and the `fst` field to the result of normalization:

```
Seq lazy_norm_in_place(Seq node) {
  switch (node.kind) {
    case Append:
      if (node.snd == null) return node.fst;
      node.fst = norm_rec(node.fst, node.snd); node.snd = null;
      return node.fst;
    default: return node;   }   }
```

Interestingly, this in-place modification is required for the *correctness* of binding of non-tail variables in patterns. The pattern matching algorithm [4] naturally supports only those pattern variables that bind to tails of sequence values; variables binding to non-tail sequences are handled by a trick. Namely, binding a non-tail variable `x` is accomplished in two stages. The first stage performs pattern matching and—as it traverses the input sequence—sets auxiliary variables $\texttt{x}_b$ and $\texttt{x}_e$ to the beginning and end of the subsequence. The second stage computes `x` from $\texttt{x}_b$ and $\texttt{x}_e$ by traversing the sequence beginning at $\texttt{x}_b$ and copying nodes until it reaches $\texttt{x}_e$. In both stages, the program traverses the same sequence, performing normalization along the way. In-place modification guarantees that during both traversals we will encounter *physically* the same concrete nodes, and so, in the second stage, we are justified in detecting the end of the subsequence by checking physical equality between the current node and $\texttt{x}_e$.

Because of creation of fresh `SeqAppend` nodes, the lazy normalization algorithm can allocate more memory than its eager counterpart. However, we can show that this results in no more than a constant factor overhead. A node is said to be a *left node* if it is pointed by the `fst` pointer of a `SeqAppend`. There are two cases when the algorithm creates a new `SeqAppend` node: when it traverses a left `SeqAppend` node, and when it reaches the leftmost concrete element. In both cases, the newly created nodes are *not* left nodes and so will not lead to further creation of `SeqAppend` nodes during subsequent normalizations. Hence, lazy normalization allocates at most twice as much memory as eager normalization.

We now present some measurements quantifying the consequences of this overhead on running time. The table below shows running times for two variants of the phone book application from Section 2, executed on an address book of 250,000 entries. (Our experimental setup is described below in Section 4.) The first variant constructs the result as in Section 2, appending to the end. The second variant constructs the result by by appending to the front:

```
res = [[ <person> n, t </>, res ]];
```

This variant favors the non-lazy tree representation from the previous subsection, which serves as a baseline for our lazy optimizations. Since our implementation recognizes prepending singleton sequences as a special case, no lazy structures are created when the second program is executed, and, consequently all concatenation approaches behave the same. For the back-appending program, the system runs out of memory using eager concatenation, while both lazy concatenation approaches perform reasonably well. Indeed, the performance of the lazy representations for the back-appending program is within 10% of the performance of the non-lazy representation for the front-appending program, which favors such a representation.

| | eager concatenations | eager normalization | lazy normalization |
|---|---|---|---|
| back appending | $\infty$ | 1,050 ms | 1,050 ms |
| front appending | 950 ms | 950 ms | 950 ms |

This comparison does not show any difference between the lazy and eager normalization approaches. We have also compared performance of eager vs. lazy normalization on the benchmarks discussed below in Section 4. Their performance is always close, with slight advantage for one or the other depending on workload. On the other hand, for programs that explore only part of a sequence, lazy normalization can be *arbitrarily* faster, making it a clear winner overall.

Our experience suggests that, in common usage patterns, our representation exhibits constant amortized time for all operations. It is possible, however, to come up with scenarios where repeatedly accessing the first element of a sequence may take linear time for each access. Consider the following program fragment:

```
[[any]] res1 = [[]];   [[any]] res2 = [[]];
while (true) {
  res1 = [[res1, <a/>]];   res2 = [[res1, <b/>]];
  match (res2) {
    case [[<(Tag x)/>, any]]: ...use x...    }  }
```

Since the pattern matching expression extracts only the first element of `res2`, only the top-level `SeqAppend` object of the sequence stored in `res2` is modified in-place during normalization. The `SeqAppend` object of the sequence stored in `res1` is not modified in-place, and, consequently, is completely renormalized during each iteration of the loop.

Kaplan, Tarjan and Okasaki [13] describe *catenable steques*, which provide all the functionality required by XTATIC pattern-matching algorithms with operations that run in constant amortized time in the presence of sharing. We have implemented their algorithms in C♯ and compared their performance with that of our representation using the lazy normalization algorithm. The steque implementation is slightly more compact—on average it requires between 1.5 and 2 times less memory than our representation. For the above tricky example, catenable steques are also fast, while XTATIC's representation fails on sufficiently large sequences. For more common patterns of operations, our representation is more efficient. The following table shows running times of a program that builds a sequence by back-appending one element at a time and fully traverses the constructed sequence. We ran the experiment for sequences of four different sizes. The implementation using catenable steques is significantly slower than our much simpler representation because of the overhead arising from the complexity of the steque data structures.

| | Steques | XTATIC |
|---|---|---|
| n = 10,000 | 70 ms | 6 ms |
| n = 20,000 | 140 ms | 12 ms |
| n = 30,000 | 230 ms | 19 ms |
| n = 40,000 | 325 ms | 31 ms |

## 4 Measurements

This section describes performance measurements comparing XTATIC with some other XML processing systems. Our goal in gathering these numbers has been to verify that our current implementation gives reasonable performance on a range of tasks and datasets, rather than to draw detailed conclusions about relative speeds of the different systems. (Differences in implementation platforms and languages, XML processing styles, etc. make the latter task well nigh impossible!)

Our tests were executed on a 2GHz Pentium 4 with 512MB of RAM running Windows XP. The XTATIC and DOM experiments were executed on Microsoft .NET version 1.1. The CDUCE interpreter (CVS version of November 25th, 2003) was compiled natively using ocamlopt 3.07+2. QIZX/OPEN and Xalan XSLTC were executed on SUN JAVA version 1.4.2. Since this paper is concerned with run-time data structures, our measurements do not include static costs of typechecking and compilation. Also, since the current implementation of XTATIC's XML parser is inefficient and does not reveal much information about the performance of our data model, we factor out parsing and loading of input XML documents from our analysis. Each measurement was obtained by running a program with given parameters ten times and averaging the results. We selected sufficiently large input documents to ensure low variance of time measurements and to make the overhead of just-in-time compilation negligible. The XTATIC

programs were compiled using the lazy append with lazy normalization policy described in Section 3.

We start by comparing XTATIC with the QIZX/OPEN [14] implementation of XQUERY. Our test is a small query named `shake` that counts the number of distinct words in the complete Shakespeare plays, represented by a collection of XML documents with combined size of 8Mb. The core of the `shake` implementation in XQUERY is a call to a function `tokenize` that splits a chunk of character data into a collection of white-space-separated words. In XTATIC, this is implemented by a generic pattern matching statement that extracts the leading word or white space, processes it, and proceeds to handle the remainder of the `pcdata`. Each time, this remainder is boxed into a `SeqSubstring` object, only to be immediately unboxed during the next iteration of the loop. We believe this superfluous manipulation is the main reason why XTATIC is more than twice slower than QIZX/OPEN in this example.

| | `shake` |
|---|---|
| XTATIC | 7,500 ms |
| QIZX/OPEN | 3,200 ms |

We also implemented several XQUERY examples from the XMark suite [15], and ran them on an 11MB data file generated by XMark (at "factor 0.1"). XTATIC substantially outperforms QIZX/OPEN on all of these benchmarks—by 500 times on `q01`, by 700 times on `q02`, by six times on `q02`, and by over a thousand times on `q08`. This huge discrepancy appears to be a consequence of two factors. Firstly, QIZX/OPEN, unlike its commercial counterpart, does not use indexing, which for examples such as `q01` and `q02` can make a dramatic performance improvement. Secondly, we are translating high-level XQUERY programs into low-level XTATIC programs—in effect, performing manual query optimization. This makes a comparison between the two systems problematic, since the result does not provide much insight about the underlying representations.

Next, we compare XTATIC with two XSLT implementations: .NET XSLT and Xalan XSLTC. The former is part of the standard C♯ library; the latter is an XSLT compiler that generates a JAVA class file from a given XSLT template.

We implemented several transformations from the XSLTMark benchmark suite [16]. The `backwards` program traverses the input document and reverses every element sequence; `identity` copies the input document; `dbonerow` searches a database of person records for a particular entry, and `reverser` reads a `PCDATA` fragment, splits it into words, and outputs a new `PCDATA` fragment in which the words are reversed. The first three programs are run on a 2MB XML document containing 10,000 top-level elements; the last program is executed on a small text fragment.

| | `backwards` | `identity` | `dbonerow` | `reverser` |
|---|---|---|---|---|
| XTATIC | 450 ms | 450 ms | 13 ms | 2.5 ms |
| .NET XSLT | 2,500 ms | 750 ms | 300 ms | 9 ms |
| Xalan XSLTC | 2,200 ms | 250 ms | 90 ms | 0.5 ms |

XTATIC exhibits equivalent speed for `backwards` and `identity` since the cost of reversing is approximately equal to the cost of copying a sequence in the presence of lazy concatenation. The corresponding XSLT programs behave differently since `backwards` is implemented by copying *and* sorting every sequence

according to the position of the elements. The XSLT implementations are relatively efficient on `identity`. This may be partially due to the fact that they use a much more compact read-only representation of XML documents. XTATIC is substantially slower than Xalan XSLTC on the `pcdata`-intensive `reverser` example. We believe the reason for this is, as in the case of `shake` in the comparison with QIZX/OPEN, the overhead of our `pcdata` implementation for performing text traversal. Conversely, XTATIC is much faster on `dbonerow`. As with QIZX/OPEN, this can be explained by the difference in the level of programming detail—a single XPATH line in the XSLTC program corresponds to a low-level XTATIC program that specifies how to search the input document efficiently.

| | `split` | `addrbook` |
|---|---|---|
| XTATIC | 950 ms | 1,050 ms |
| CDUCE | 650 ms | 1,300 ms |

In the next pair of experiments, we compare XTATIC with CDUCE [7] on two programs: `addrbook` and `split`. The first of these was introduced in Section 2 (the CDUCE version was coded to mimick the XTATIC version, i.e. we did not use CDUCE's higher-level `transform` primitive); it is run on a 25MB data file containing 250,000 `APers` elements. The second program traverses a 5MB XML document containing information about people and sorts the children of each person according to gender. Although it is difficult to compare programs executed in different run-time frameworks and written in different source languages, we can say that, to a rough first approximation, XTATIC and CDUCE exhibit comparable performance. An important advantage of CDUCE is a very memory-efficient representation of sequences. This is compensated by the fact that XTATIC programs are (just-in-time) compiled while CDUCE programs are interpreted.

The next experiment compares XTATIC with XACT [17]. We use two programs that are part of the XACT distribution—`recipe` processes a database of recipes and outputs its HTML presentation; `sortedaddrbook` is a version of the address book program introduced in Section 2 that sorts the output entries. We ran `recipe` on a file containing 525 recipes and `sortedaddrbook` on a 10,000 entry address book.(Because of problems installing XACT under Windows, unlike the other experiments, comparisons with XACT were executed on a 1GHz Pentium III with 256MB of RAM running Linux.) For both programs XTATIC is substantially faster. As with XQUERY, this comparison is not precise because of a mismatch between XML processing mechanisms of XTATIC and XACT. In particular, the large discrepancy in the case of `recipe` can be partly attributed to the fact that its style of processing in which the whole document is traversed and completely rebuilt in a different form is foreign to the relatively high level XML manipulation primitives of XACT but is quite natural to the relatively low level constructs of XTATIC.

| | `recipe` | `sortedaddrbook` |
|---|---|---|
| XTATIC | 250 ms | 1,600 ms |
| XACT | 60,000 ms | 10,000 ms |

The last experiment compares XTATIC with a C♯ program using DOM and the .NET XPATH library, again using the `addrbook` example on the 25MB input file. The C♯ program employs

| | `addrbook` |
|---|---|
| XTATIC | 1,050 ms |
| DOM/Xpath | 5,100 ms |

XPath to extract all the `APers` elements with `tel` children, destructively removes their `email` children, and returns the obtained result. This experiment confirms that DOM is not very well-suited for the kind of functional manipulation of sequences prevalent in Xtatic. The DOM data model is geared for destructive modification and random access traversal of elements and, as a result, is much more heavyweight.

## 5 Related Work

We have concentrated here on the runtime representation issues that we addressed while building an implementation of Xtatic that is both efficient and tightly integrated with C♯. Other aspects of the Xtatic design and implementation are described in several companion papers—one surveying the most significant issues faced during the design of the language [11], another presenting the core language design, integrating the object and tree data models and establishing basic soundness results [3], and the third proposing a technique for compiling regular patterns based on *matching automata* [9].

There is considerable current research and development activity aimed at providing convenient support for XML processing in both general-purpose and domain-specific languages. In the latter category, XQuery [6] and XSLT [5] are special-purpose XML processing languages specified by W3C that have strong industrial support, including a variety of implementations and wide user base. In the former, the CDuce language of Benzaken, Castagna, and Frisch [7] generalizes XDuce's type system with intersection and function types. The Xen language of Meijer, Schulte, and Bierman [18] is a proposal to significantly modify the core design of C♯ in order to integrate support for objects, relations, and XML (in particular, XML itself simply becomes a syntax for serialized object instances). Xact [17, 8] extends Java with XML processing, proposing an elegant programming idiom: the creation of XML values is done using XML templates, which are immutable first-class structures representing XML with named gaps that may be filled to obtain ordinary XML trees. XJ [19] is another extension of Java for native XML processing that uses W3C Schema as a type system and XPath as a navigation language for XML. XOBE [20] is a source to source compiler for an extension of Java that, from language design point of view, is very similar to Xtatic. Scala is a developing general-purpose web services language that compiles into Java bytecode; it is currently being extended with XML support [21].

So far, most of the above projects have concentrated on developing basic language designs; there is little published work on serious implementations. (Even for XQuery and XSLT, we have been unable to find detailed descriptions of their run-time representations.) We summarize here the available information.

Considerable effort, briefly sketched in [7], has been put into making the CDuce's OCaml-based interpreter efficient. They address similar issues of text and tree representations and use similar solutions. CDuce's user-visible datatype for strings is also the character list, and they also implement its optimized

alternatives—the one described in the paper resembles our `SeqSubstring`. CDuce uses lazy list concatenation, but apparently only with eager normalization. Another difference is the object-oriented flavor of our representations.

Xact's implementation, developed independently and in parallel with Xtatic but driven by similar needs (supporting efficient sharing, etc.) and targeting a similar (object-oriented) runtime environment, has strong similarities to ours; in particular, lazy data structures are used to support efficient gap plugging. Our preliminary performance measurements may be viewed as validating the representation choices of both implementations. Xtatic's special treatment of `pcdata` does not appear to be used in Xact.

The current implementations of XOBE and XJ are based on DOM, although the designs are amenable to alternative back-ends.

Kay [22] describes the implementation of Version 6.1 of his XSLT processor Saxon. The processor is implemented in Java and, like in our approach, does not rely on a pre-existing Java DOM library for XML data representation, since DOM is again too heavyweight for the task at hand: e.g. it carries information unnecessary for XPath and XSLT (like entity nodes) and supports updates. Saxon comes with two variants of run time structures. One is object-oriented and is similar in spirit to ours. Another represents tree information as arrays of integers, creating node objects only on demand and destroying them after use. This model is reportedly more memory efficient and quicker to build, at the cost of slightly slower tree navigation. Overall, it appears to perform better and is provided as the default in Saxon.

In the broader context of functional language implementations, efficient support for list (and string) concatenation has long been recognized as an important issue. An early paper by Morris, Schmidt and Wadler [23] describes a technique similar to our eager normalization in their string processing language Poplar. Sleep and Holmström [24] propose a modification to a lazy evaluator that corresponds to our lazy normalization. Keller [25] suggests using a lazy representation without normalization at all, which behaves similarly to database B-trees, but without balancing. We are not aware of prior studies comparing the lazy and eager alternatives, as we have done here.

More recently, the algorithmic problem of efficient representation for lists with concatenation has been studied in detail by Kaplan, Tarjan and Okasaki [13]. They describe *catenable steques* which support constant amortized time sequence operations. We opted for the simpler representations described here out of concern for excessive constant factors in running time arising from the complexity of their data structures (see Section 3.2.)

Another line of work, started by Hughes [26] and continued by Wadler [27] and more recently Voigtlander [28] considers how certain uses of list concatenation (and similar operations) in an applicative program can be eliminated by a systematic program transformation, sometimes resulting in improved asymptotic running times. In particular, these techniques capture the well-known transformation from the quadratic to the linear version of the reverse function. It is not clear, however, whether the techniques are applicable outside the pure functional

language setting: e.g., they transform a recursive function $f$ that uses append to a function $f'$ that uses only list construction, while in our setting problematic uses of append often occur inside imperative loops.

Prolog's difference lists [29] is a logic programming solution to constant time list concatenation. Using this technique requires transforming programs operating on regular lists into programs operating on difference lists. This is not always possible. Marriott and Søndergaard [30] introduce a dataflow analysis that determines whether such transformation is achievable and define the automatic transformation algorithm. We leave a more detailed comparison of our lazy concatenation approach and the difference list approach for future work.

## Acknowledgements

Parts of the XTATIC compiler were implemented by Eijiro Sumii and Stephen Tse. Conversations with Eijiro contributed many ideas to XTATIC and this paper. We also thank Haruo Hosoya, Alain Frisch, Christian Kirkegaard, and Xavier Franc for discussing various aspects of this work. Our work on XTATIC has been supported by the National Science Foundation under Career grant CCR-9701826 and ITR CCR-0219945, and by gifts from Microsoft.

## References

1. Hosoya, H., Pierce, B.C.: XDuce: A statically typed XML processing language. ACM Transactions on Internet Technology **3** (2003) 117–148
2. Hosoya, H., Vouillon, J., Pierce, B.C.: Regular expression types for XML. In: Proceedings of the International Conference on Functional Programming (ICFP). (2000)
3. Gapeyev, V., Pierce, B.C.: Regular object types. In: European Conference on Object-Oriented Programming (ECOOP), Darmstadt, Germany. (2003) A preliminary version was presented at FOOL '03.
4. Hosoya, H., Pierce, B.C.: 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.
5. W3C: XSL Transformations (XSLT) (1999) `http://www.w3.org/TR/xslt`.
6. : XQuery 1.0: An XML Query Language, W3C Working Draft (2004) `http://www.w3.org/TR/xquery/`.
7. Benzaken, V., Castagna, G., Frisch, A.: CDuce: An XML-centric general-purpose language. In: ACM SIGPLAN International Conference on Functional Programming (ICFP), Uppsala, Sweden. (2003) 51–63
8. Christensen, A.S., Kirkegaard, C., Møller, A.: A runtime system for XML transformations in Java. In Bellahsène, Z., Milo, T., Michael Rys, e.a., eds.: Database and XML Technologies: International XML Database Symposium (XSym). Volume 3186 of Lecture Notes in Computer Science., Springer (2004) 143–157
9. Levin, M.Y.: Compiling regular patterns. In: ACM SIGPLAN International Conference on Functional Programming (ICFP), Uppsala, Sweden. (2003)

10. Gapeyev, V., Levin, M.Y., Pierce, B.C., Schmitt, A.: XML goes native: Runtime representations for Xtatic. Technical Report MS-CIS-04-23, University of Pennsylvania (2004)
11. Gapeyev, V., Levin, M.Y., Pierce, B.C., Schmitt, A.: The Xtatic experience. Technical Report MS-CIS-04-24, University of Pennsylvania (2004)
12. Tabuchi, N., Sumii, E., Yonezawa, A.: Regular expression types for strings in a text processing language. In den Bussche, J.V., Vianu, V., eds.: Proceedings of Workshop on Types in Programming (TIP). (2002) 1–18
13. Kaplan, H., Okasaki, C., Tarjan, R.E.: Simple confluently persistent catenable lists. SIAM Journal on Computing **30** (2000) 965–977
14. Franc, X.: Qizx. `http://www.xfra.net/qizxopen` (2003)
15. Schmidt, A.R., Waas, F., Kersten, M.L., Carey, M.J., Manolescu, I., Busse, R.: XMark: A benchmark for XML data management. In: Proceedings of the International Conference on Very Large Data Bases (VLDB), Hong Kong, China (2002) 974–985 See also `http://www.xml-benchmark.org/`.
16. DataPower Technology, Inc.: XSLTMark. `http://www.datapower.com/xml_community/xsltmark.html` (2001)
17. Kirkegaard, C., Møller, A., Schwartzbach, M.I.: Static analysis of XML transformations in Java. IEEE Transactions on Software Engineering **30** (2004) 181–192
18. Meijer, E., Schulte, W., Bierman, G.: Programming with circles, triangles and rectangles. In: XML Conference and Exposition. (2003)
19. Harren, M., Raghavachari, B.M., Shmueli, O., Burke, M., Sarkar, V., Bordawekar, R.: XJ: Integration of XML processing into Java. Technical Report rc23007, IBM Research (2003)
20. Kempa, M., Linnemann, V.: On XML objects. In: Workshop on Programming Language Technologies for XML (PLAN-X). (2003)
21. Emir, B.: Extending pattern matching with regular tree expressions for XML processing in Scala. Diploma thesis, EPFL, Lausanne; `http://lamp.epfl.ch/~buraq` (2003)
22. Kay, M.H.: Saxon: Anatomy of an xslt processor (2001) `http://www-106.ibm.com/developerworks/library/x-xslt2/`.
23. Morris, J.H., Schmidt, E., Wadler, P.: Experience with an applicative string processing language. In: ACM Symposium on Principles of Programming Languages (POPL), Las Vegas, Nevada. (1980) 32–46
24. Sleep, M.R., Holmström, S.: A short note concerning lazy reduction rules for append. Software Practice and Experience **12** (1982) 1082–4
25. Keller, R.M.: Divide and CONCer: Data structuring in applicative multiprocessing systems. In: Proceedings of the 1980 ACM conference on LISP and functional programming. (1980) 196–202
26. Hughes, J.: A novel representation of lists and its application to the function "reverse". Information Processing Letters **22** (1986) 141–144
27. Wadler, P.: The concatenate vanishes. Note, University of Glasgow (1987) (revised 1989).
28. Voigtländer, J.: Concatenate, reverse and map vanish for free. In: ACM SIGPLAN International Conference on Functional Programming (ICFP), Pittsburgh, Pennsylvania. (2002) 14–25
29. Sterling, L., Shapiro, E.: The Art of Prolog. MIT Press (1986)
30. Marriott, K., Søndergaard, H.: Difference-list transformation for prolog. New Generation Computing **11** (1993) 125–157