XML Goes Native: Run-time Representations for Xtatic

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

Technical Report MS-CIS-04-23
Department of Computer and Information Science
University of Pennsylvania

October 26, 2004

Abstract

Xtatic is a lightweight extension of C$^\sharp$ offering native support for statically typed XML processing. XML trees are built-in values in Xtatic, and static analysis of the trees created and 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$^\sharp$. 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 [14, 15, 16], 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 statically typechecking computations involving XML, and a powerful form of pattern matching called *regular patterns*. The goals of the Xtatic project are, first, to bring these technologies to a wide audience by integrating them with a mainstream object-oriented language and, second, to demonstrate an implementation with good performance. We use C$^\sharp$ 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$^\sharp$ is accomplished by two steps. First, the subtype hierarchy of tree types from XDuce is grafted into the C$^\sharp$ 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 [9]. Algorithms for checking subtyping and inferring types for variables bound in patterns can be adapted straightforwardly from those of XDuce ([16] and [14]).

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 [37], XQuery [39], CDuce [1], and Xact [24, 2], 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$^\sharp$. One major function of this compiler is to translate the high-level pattern matching statements of Xtatic into low-level C$^\sharp$ code that is efficient and compact. A previous paper [25] 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:

- 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;
- 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);
- 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
- a proxy scheme allowing foreign XML representations such as DOM to be manipulated by Xtatic programs without first translating them to our representation.

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. Section 3 discusses the translation scheme chosen for the compilation of Xtatic types. The heart of the paper is Sections 4 and 5, which describe and evaluate our representations for trees and textual data. Section 6 deals with issues arising at the interface of Xtatic and pure C$^\sharp$. Section 7 summarizes results of benchmarking programs compiled by Xtatic against some other popular XML processing tools. Section 8 discusses related work; Section 9, future work.

2 Language Overview

This section sketches just the aspects of the Xtatic design that directly impact runtime representation issues. A full description of the language can be found in [8, 9].

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$^\sharp$, 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. For example, in the presence of these definitions

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

our address book could be given the type `APers*`.

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$^\sharp$ 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; }
```

An XTATIC pattern is actually just a regular type decorated with variable binders; the run-time behavior of pattern matching is just regular tree language membership testing plus subtree (and sub-sequence) extraction.

The XTATIC type system ensures that each pattern match is complete (some branch will always succeed). The compiler can also infer types for variables bound by patterns. For instance, in the first `case` above, the variable `t` is given the type `<tel>pcdata</>`, even though its associated sub-pattern matches the more general type `<tel>any</>`, the type "`any`" being the least precise sequence type. Similarly, the rest of the sequence is matched against the type `any`, but may be stored into the variable `ps`, which has type `APers*`, as the type checker infers its type precisely.

Compared with the facilities available in pure C$^\sharp$(such as the raw DOM API), regular pattern matching allows much cleaner and more readable implementations of many tree investigation and transformation algorithms. However, compared with other native XML processing languages, XTATIC's pattern matching primitives are still fairly low-level: for example, no special syntax is provided for collecting *all* sub-trees matching a given pattern, or for iterating over sequences. We are currently investigating how best to add more powerful pattern matching; for now, our implementation efforts are concentrated on achieving good performance for low-level XML processing code.

The integration of XML sequences with C$^\sharp$ 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`. This approach is justified by the homogeneous translation used by our compiler. 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$^\sharp$ `char` class[1] and taking `pcdata` to be an abbreviation for `<(char)/>*`. In the concrete syntax, we write `‘foo‘` for the sequence type `<(char`$_f$`)/><(char`$_o$`)/><(char`$_o$`)/>` 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
```

[1]These singleton classes do not actually correspond to anything in the C$^\sharp$ runtime, where the class `char` cannot have any subclasses. They are compiled away by the XTATIC compiler.

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 [35]. For instance, the Xtatic type `‘a‘,pcdata,‘b‘` corresponds to the regular expression `a.*b`.

The current Xtatic design adopts a very simple, untyped view of the attributes attached to XML elements: attributes may appear in tree values and in regular patterns, but they are not tracked by the type system. This is an interim solution: a language with Xtatic's goals should clearly treat attributes statically. At the moment, however, a fully satisfactory type system for attributes remains a matter of ongoing research (designs based on conventional record types, such as CDuce's, do not easily support dynamic transformation of documents with unknown or partially known types; Hosoya and Murata [13] have a proposal with most of the properties that we want, but of daunting complexity).

3 Translation Scheme

The most fundamental design choice for compiling a language like Xtatic is whether to use a heterogeneous translation scheme, where different XML types from the source language can be translated to different target language types, or a homogeneous scheme, in which all XML types are represented using a single target language type. This section discusses both alternatives and explains why we chose the latter for the current compiler implementation.

To begin with, it is useful to distinguish both of these schemes from a third popular alternative, commonly called *data binding.* A data binding for XML can be described as (1) a mapping $[\![—]\!]$ from XML types (in some XML schema language) to target language types, and (2) an injective mapping $[\![—]\!]_{\_\_}$ from XML values (indexed by their types) to target language values, such that $[\![\mathtt{t}]\!]_{\mathtt{T}} \in [\![\mathtt{T}]\!]$ for each XML value `t` of type `T`. That is, a data binding is a way of embedding XML types and values into the types and values of the target language. (Note that we are only interested in mapping values and types—there is no source language as such; the purpose of a data binding is to allow XML values to be "imported" as ordinary target language data structures and manipulated by writing ordinary target language code.) Several popular data bindings are available, mapping XML to a variety of host languages [34, etc.]. However, a data binding cannot form the basis of a compiler for an XML-oriented source language unless it also preserves the subtyping relation of the source language—i.e., if `S` and `T` are XML types with `S <: T`, we should have $[\![\mathtt{S}]\!]$ `<:` $[\![\mathtt{T}]\!]$ in the target language. Data bindings are generally designed so that *some* subtype relationships between source types are preserved by translation, but usually—and in particular for Xtatic and C$^\sharp$—it is not possible to reflect source subtyping completely.[2]

A *heterogeneous translation scheme* may be thought of as a data binding plus a way of translating source-language subtyping into *coercion functions* in the target language. Essentially, for every pair of source types `S` and `T` with `S <: T`, there is a coercion function $[\![\mathtt{S} <: \mathtt{T}]\!] \in [\![\mathtt{S}]\!] \longrightarrow [\![\mathtt{T}]\!]$. Sometimes (when we already have $[\![\mathtt{S}]\!]$ `<:` $[\![\mathtt{T}]\!]$ in the target language) these coercions are just identity functions; but in other cases they perform real, run-time changes of representation. The coercion functions are inserted into target code by the compiler wherever it sees that an (implicit or explicit) upcast is being used during typechecking.

Heterogeneous translation schemes are attractive because they allow specialized representations that can be accessed efficiently, avoiding many run-time tests and making compact use of memory (see, e.g., [6]). However, they raise some tricky design challenges, which we have not attempted to address in Xtatic. In particular, because performing run-time representation changes on large XML structures could be prohibitively expensive, the subtype relation and representation must be carefully designed together so that coercions will usually only be applied to small structures.

Instead, for the current Xtatic compiler, we choose the more straightforward (and more common) approach of using a *homogeneous* translation, translating all XML types to a single target type. In the case of Xtatic, there is a natural candidate for this type: the special class `Seq`, which is defined to be a supertype

[2]Indeed, it is in the nature of data bindings that this property will generally fail: if the target language type system were flexible enough to encode all the desired relationships between XML types, then there would be little need (concrete syntax aside) to have a separate language of XML types in the first place!

of all XML types. In effect, the compilation process takes the rich regular type structure hanging below the `Seq` class and squashes it all down to just `Seq`.

The homogeneous translation scheme also has some potential costs, which we will need to deal with below. In particular, when type information is needed for a runtime check (e.g., if we have a structure of type `Seq` and we wish to cast it to some more precise regular type), a naive implementation might need to walk over a large structure to recalculate its type; we show in Section 6.2 how this can be avoided by judicious (programmer-controlled) tagging of trees with their types. Another issue is that the precise compile-time type information that is thrown away by the homogeneous translation needs to be stored on the side somewhere so that cross-compilation-unit overloading can be resolved properly; this point is discussed in Section 6.1.

4 Representing Trees

We now turn to the design of efficient representations for XML trees. First, we select a tag representation that supports separate compilation and XML namespaces (Section 4.1). Next, we design a tree representation that supports XTATIC's view of trees as shared and immutable structures (Section 4.2). 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 4.3). Finally, XTATIC needs to inter-operate with other XML representations available in .NET, in particular DOM. We show how DOM structures can masquerade as instances of our XTATIC trees in a type-safe manner (Section 4.4).

4.1 Tags

Our implementation defines a class `Tag`, and every particular XML tag is an *object* of this class. 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. Separate compilation is supported by allocating these tags at start-up time: each separately compiled library adds its tags to a common hash table when loaded. Hence, every library associates the same object to a given tag. Moreover, this tag representation simplifies the recovery of a tag name, needed to print it.

We considered several other run-time representations of tags. One, directly corresponding to the formal definition of the XTATIC data model [9], encodes XML tags by different *classes*. This approach does not work in the context of separate compilation since the same XML tag occurring in different compilation units would be mapped to *distinct* classes sharing the same name but residing in different assemblies. Representing tags by values of an enumeration type offers the ability to compile pattern-matching into efficient `switch` statements, but, like the class-based approach, does not work with separate compilation, since we cannot guarantee that the same tag will correspond to the same enumeration value in every compilation unit. One might also represent an XML tag by a single string containing both the namespace and the tag name separated by some special character, and hash this string using a fixed function carefully chosen as to minimize collisions. This approach, similar to the one used for the implementation of labels in OCaml [11], is not applicable in our setting as the name of the tag is no longer available at runtime.

4.2 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, such as pattern matching, 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 (which should be implemented by a good .NET

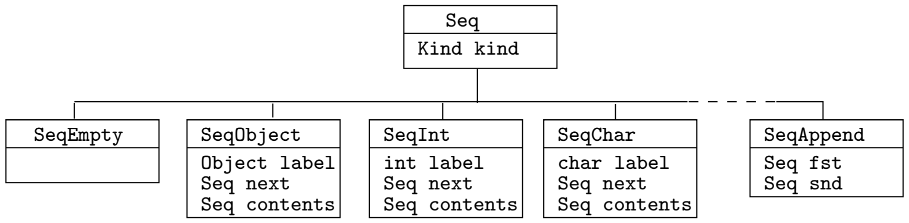


Figure 1: Classes used for representing sequences.

JIT compiler using a jump table) 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$^\sharp$ 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.

Briefly, 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`.

4.3 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:[3] 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.

[3]The problem of efficient list concatenation has, of course, been studied in the functional programming community, and a number of techniques have been proposed; see Section 8. We describe here our adaptation of these ideas to the specifics of XTATIC—for example, in-place updates will turn out to be critical for the correctness of pattern variable binding.

```
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, ... */   }  }
```

Figure 2: Lazy Normalization Algorithm.

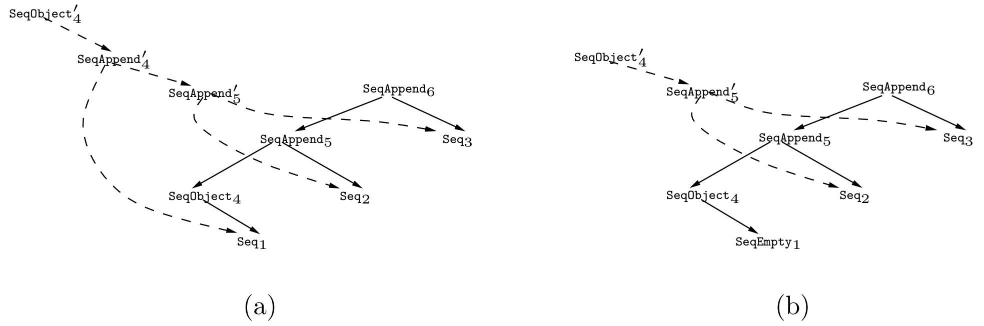


Figure 3: Lazy normalization of lazy sequences. In (a), the leftmost concrete element has a right sibling; in (b) it does not. Dotted pointers and their source objects are created during normalization.

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 (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 [14] 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

	eager concatenations	eager normalization	lazy normalization
back appending	∞	1,050 ms	1,050 ms
front appending	950 ms	950 ms	950 ms

Figure 4: Running times for two variants of the phone book application.

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 $\mathtt{x}_b$ and $\mathtt{x}_e$ to the beginning and end of the subsequence. The second stage computes `x` from $\mathtt{x}_b$ and $\mathtt{x}_e$ by traversing the sequence beginning at $\mathtt{x}_b$ and copying nodes until it reaches $\mathtt{x}_e$. 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 $\mathtt{x}_e$.

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, as follows. 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. Figure 4 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 7.) 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.

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 7. 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 [19, 20, 30, 18] 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$^\sharp$ 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, however, 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.

	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

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.

4.4 DOM Interoperability

XTATIC modules are expected to be useful in applications built in other .NET languages with the use of extensive .NET libraries. The latter already contain support for XML, collected in the `System.Xml` namespace. It can greatly enhance the usefulness of XTATIC if its XML manipulation facilities can be applied to native .NET XML representations and, conversely, if XTATIC XML data can be accessed from vanilla C$^\sharp$ code. We have explored the former direction of this two-sided interoperability problem by implementing support for DOM, one popular XML representation available in .NET.

A straightforward solution for accessing DOM from XTATIC would be to translate any DOM data of interest into our representation in its entirety. This is wasteful, however, if an XTATIC program ends up accessing only a small portion of the document. A better idea is to wrap a DOM fragment in a lazy structure (using another subclass, called `SeqDom`, of `Seq`) and investigate its contents only as needed during pattern matching. However, since DOM structures are mutable, we need to take some care to maintain type safety. We do this by investigating the underlying DOM structure just once and copying the parts we have seen into immutable `Seq` nodes.

Concretely, a `SeqDom` object has two fields, `dom` and `seq`, one of which is always `null`. When a `SeqDom` object is created, its `dom` field points to a DOM element node, meaning that the object represents the XML fragment consisting of the DOM node, its following siblings, and its children nodes. (The DOM element node's pointers to its parent and previous siblings are not relevant for the meaning a `SeqDom` wrapper, since XTATIC never needs to traverse them.)

The actual inspection of the underlying DOM nodes happens during normalization. In the most common case, normalization of a `SeqDom` object considers its underlying DOM element node (call it `e`) and creates a new `SeqObject` object with the `label` field corresponding to `e.Name` and fields `contents` and `next` pointing to newly created `SeqDom` objects corresponding to the DOM nodes `e.FirstChild` and `e.NextSibling`. In cases where either of the latter two DOM nodes is not an element node—i.e. it is either `null` or a DOM text node—the normalization transforms it directly to a `SeqEmpty` or an appropriate XTATIC `pcdata` representation. Finally, the normalization modifies the original `SeqDom` object by setting the `dom` field to `null` and pointing the `seq` field to the newly created `SeqObject`.

A `SeqDom` object can be initially obtained by applying a special library function to a DOM node. The return type of this function is `[[xml]]`, the least precise XML type. A successful pattern match of such a value against a pattern more detailed than `[[xml]]` has the side-effect of transforming, via the normalizations that get invoked along the way, some initial "spine" of the value into our native XTATIC representation. If the pattern contains subpatterns typed as `[[xml]]`, the corresponding value fragments may safely be put in `SeqDom` wrappers; it does not matter if the underlying DOM structures are later modified, since no assumptions about their actual type have yet been made by XTATIC code.

As a consequence, the parts of the original DOM structure may be destructively updated at any time after it was wrapped in `DomSeq`. The results of such updates are visible to XTATIC only if they happen before pattern matching reaches them. (Caveat: This statement applies to a single-threaded context. It remains true in a multithreaded setup only when methods of the DOM implementation are thread safe—something that .NET DOM implementation does not guarantee.)

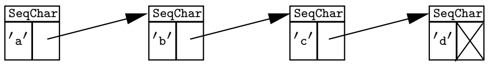
Using `SeqDom` wrappers in the context of lazy concatenation with subsequent normalization of the resulting structures into concrete XTATIC sequences is crucial to the efficiency of our approach. An alternative design that implemented concatenation of DOM wrappers as a DOM wrapper and at the same time wanted to support shared-structure view of data would have to do extensive deep cloning of DOM fragments. Otherwise, the doubly-linked nature of DOM structures could lead to unintended sharing violations.

We plan to address the other direction of the interoperability problem—efficiently *exporting* XML trees created by XTATIC for use in native C$^\sharp$ code—by implementing one of the `System.Xml` access interfaces on top of XTATIC sequences.

5 Representing Text

In this section we describe several ways of representing `pcdata` and weigh the merits of each approach.

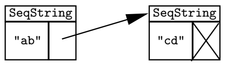
The definition of the type `pcdata` as `<(char)/>*` immediately suggests a naive representation using linked lists of `SeqChar` objects. For example, the text `‘abcd‘` would be represented as:



The primary advantage of this representation is that we can directly use our sequence representation and tree pattern matching algorithms to inspect textual data. The primary disadvantage is also obvious: extremely inefficient use of memory, as each character is represented by a `SeqChar` object.

A more compact alternative is to use native C$^\sharp$ strings for representing `pcdata`. We have explored several variants.

The simplest of these is to extend the `Seq` class hierarchy with class `SeqString`, whose objects store a `string` value plus a pointer to the next `Seq` object. Here is one possible representation for the text `'abcd'`:



As the example suggests, a single `SeqString` node does not have to encapsulate an entire run of consecutive text. In the case where characters are added to the sequence one by one, this actually results in memory usage less efficient than the naive `SeqChar` representation, as a one-character string takes more space than a single `char`. Moreover, the following program illustrates a common case where the new representation results in worse behavior than the naive representation. It scans a given chunk of text character by character and replaces every occurrence of consecutive `'a'`s by a `'b'`.

```
[[pcdata]] process([[pcdata]] txt) {
  [[pcdata]] res = [[]];
  while (true) {
    match (txt) {
      case [[]]:
```

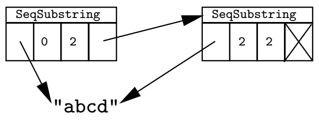
	SeqChar	SeqString	SeqSubstring	Adaptive
process	47 Mb / 2,500 ms	∞	52 Mb / 4,800 ms	44 Mb / 4,500 ms
addrbook	249 Mb / 250 ms	77 Mb / 180 ms	68 Mb / 180 ms	68 Mb / 180 ms

Figure 5: Comparison of `pcdata` representations.

```
          return res;
        case [['a'+, any rest]]:
          res = [[res, 'b']];
          txt = rest;
        case [[<(char)/> x, any rest]]:
          res = [[res, x]];
          txt = rest;
  } } }
```

In each iteration of the loop, the content of `rest` is a string that needs to be extracted—hence copied—from the current value of `txt`, resulting in quadratic space and time behavior. For this program, the naive `SeqChar` implementation does not need to allocate any new sequences for `rest`.

We can avoid this problem by refining the `SeqString` representation to point to a shared string buffer and maintain a starting offset and length. The new `pcdata` representation is encoded by class `SeqSubstring` as illustrated in the following figure.



There are cases where the `SeqSubstring` representation is less efficient than the `SeqString` scheme because of the two extra memory slots. Therefore, it is advantageous to have the three schemes coexist and choose adaptively (at instantiation time) which one to use: `SeqChar` when we find ourselves appending a single character to a sequence, `SeqString` when we append a whole string, and `SeqSubstring` when we append a piece extracted from an existing string.

An interesting side effect of the string representation is that it allows us to use the .NET regular expression library `System.Text.RegularExpressions` for pattern matching over pure PCDATA. Some advantages for this choice are ease of implementation, leveraging the highly optimized .NET library (which, for example, translates regular expressions into CLR bytecodes at run time and JIT-compiles them), and the size reduction of XTATIC-generated C$^\sharp$ code that results from delegating to library calls all the character matching code that otherwise would have to be generated. One drawback is that it requires that lazy sequences of characters be coalesced into a single `string` object that can be passed to the regular expression engine. When a string is needed for .NET regular expression matching, the sequence is (eagerly) normalized to eliminate `SeqAppends` up to the first non-character representing item and the resulting list of strings is concatenated.

We conclude this section with performance measurements from two experiments (Figure 5). The first is the program `process` given earlier in this section, evaluated on a 0.5 Mb `pcdata` file; the second is the `addrbook` example run on an XML document containing 250,000 `APers` elements. In both experiments, we measure the memory usage and processing time of the program. In the first experiment, we take a memory checkpoint right after `process` finished its work and built the result; in the second, right after the input document is loaded in memory.

In the first experiment, the `SeqString` representation did not complete because of its quadratic behavior during suffix extraction. At the time when we take the memory measurement, the document is fully fragmented and each character is boxed in a `SeqChar` or `SeqSubstring` object. Since the latter is larger, the `SeqChar` representation is more compact than the `SeqSubstring` representation for this program. The `SeqChar` scheme is also faster. The main reason of this is that suffix extraction does not perform allocation whereas in the `SeqSubstring` representation, a suffix is obtained by creating a new `SeqSubstring` object.

To see how performance of XTATIC's pattern matching over `pcdata` compares with performance of native string manipulation, we handcoded a C♯ program that implements the behavior of the `process` example. Instead of scanning each character of the input string, this program goes directly to the next substring matching `a+` and concatenates the intermediate substrings using C♯'s `StringBuilder`. This program completes in 60 ms and uses 3.5 Mb. This shows that XTATIC pays a heavy price for treating `pcdata` generically and doing a lot of boxing and for forcing character for character traversal.

The results of the second experiment contrast with the first. Since `addrbook` does not pattern match over `pcdata`, no fragmentation takes place, and we can benefit from a compact `pcdata` representation. As expected, the `SeqChar` scheme proves to be substantially less efficient than the others. Observe that `SeqSubstring` is more memory efficient than `SeqString` even though `SeqString` objects are smaller. The reason of this is that with `SeqSubstring`, we load all `pcdata` chunks into a single string buffer and avoid creating a large number of string objects.

The approaches presented in this section are in no way exhaustive. We are planning to extend the XTATIC pattern matching algorithm to experiment with several strategies to directly match strings and to compare these with the native .NET regular expression approach.

6 Calling Hell From Heaven

The homogeneous translation scheme raises some issues related to calls between XTATIC and C♯. One is static in nature and is concerned with overloaded methods whose signatures are different in XTATIC but are mapped to indistinguishable C♯ signatures. Another is a dynamic issue addressing efficient retrieval of XTATIC values from homogeneous C♯ containers. We consider these issues in turn.

6.1 Method Overloading

It is natural for XTATIC to extend C♯ method overloading to support method signatures that contain regular types. This conflicts somewhat with our homogeneous translation scheme, which would translate all such types to `Seq`, possibly resulting in shortage of signatures that could differentiate the original methods.

We resolve this problem by generating new names for all methods having arguments of types more precise than `Seq` (or, equivalently, `[[any]]`). (The renaming scheme, however, has to take into account method signatures so that, e.g., an overriding method receives the same modified name as the method it overrides). As it follows, this name mangling is not applied to methods whose regular type arguments are all of class `Seq`—this is the case that can be handled by C♯ overloading itself.

Similar treatment, for identical safety reasons, is applied to class fields.

This approach fits well with the needs of separate compilation, when a pre-compiled XTATIC library is to be used for programming larger applications, either using XTATIC or pure C♯:

- In addition to translating XTATIC code to C♯ our compiler also preserves in a separate structure signatures of methods with regular argument types.[4] Then, an XTATIC library consists of this information together with an assembly generated by a C♯ compiler.
- When an XTATIC program is compiled against the library, the preserved signature information is used to resolve overloaded method calls into appropriate mangled method names.
- A C♯ program compiled against the XTATIC library is expected to refer only to non-mangled method names,[5] that is method names with regular type arguments at most as precise as `Seq`.

[4]Currently this structure is a separate file, but we explore if the corresponding information can be stored directly in .NET assemblies.

[5]Technically, it is possible to violate safety by finding out the precise mangled names of methods. We are not aware, however, about protection for mangled methods more substantial than security through obscurity.

Consequently, if a creator of an Xtatic library needs to expose to pure C♯ code a method operating on values of a regular type, say `[[Person]]`, he needs to explicitly create a method accepting `Seq` values, casting them to `[[Person]]`, and then performing the intended functionality.

The latter scenario can be contrasted with an entirely different (hypothetical) support for method overloading that would translate a method with regular type arguments to a method with `Seq` arguments, automatically generating appropriate casting code. We believe that our simpler solution, by exposing to the programmer need to perform potentially expensive cast operations, can result in better program designs, as well as more customized error reporting and recovery.

6.2 Fast Downcasting

Part of the appeal of Xtatic is that it allows programmers to use familiar C♯ libraries to store and manipulate XML values; in particular, XML values can be stored in generic collections such as `Hashtable` and `Stack`. However, extracting values from such containers requires a downcast from `object` to the intended type of the value. In pure C♯ this operation incurs only a small time overhead, but in Xtatic, downcasting to a regular type may involve an expensive structural traversal of the entire value. To avoid this overhead, we need a way to *stamp* sequence values with a representation of their type and perform a run-time type comparison rather than full re-validation during downcasting. Our design places this stamping under programmer control.

We begin by extending the source language with a stamping construct, written `<[[T]]>e` ("stamp `e` with regular type `T`"). An expression of this form is well typed if `e` has static type `T`; in this case, the result type of the whole expression is `object`. At run-time, stamped sequences are represented by objects of class `StampedSeq`, with two fields: `stamp`, of type `Typestamp`, and `contents`, of type `Seq`. (Giving stamped values type `object` ensures that, at run-time, such values will never appear as part of other sequences. This makes sequence operations such as concatenation, normalization, and prefix extraction simpler and more efficient.)

For each regular type `T` appearing in a stamp or cast expression in the program, the compiler generates initialization code that hashes `T` type to an object of class `Typestamp`.

A cast expression of the form `([[T]])e` is executed by first checking whether the value of `e` has class `StampedSeq`; if so, it extracts the stamp, checks whether it is identical to the hash of `T`, and returns the sequence stored in the `contents` field of the `StampedSeq` object; if this fails, it falls back to the general pattern-matching algorithm, which dynamically re-validates the value.

A small experiment demonstrates the benefits of type-stamping. Consider an obfuscated program for reversing sequences belonging to the type `APers*` introduced in Section 2. We traverse the sequence and put each element in a C♯ queue. Then we dequeue one element at a time, cast it to `APers`, and add it to the result. We ran this program on two XML documents, each containing 30,000 `APers` elements. In the first document, each `APers` element has exactly one `email` child, in the second, twenty. For each document we tried two versions of the program: one with type-stamping, and one without.

	without type-stamping	with type-stamping
1 email	33 ms	28 ms
20 emails	89 ms	28 ms

On the document with single `emails`, type stamping yields only a small performance improvement, since the overhead of adding and checking the type stamps is roughly equivalent to the cost of pattern-matching a small `APers` element. In the other case, the benefits of type-stamping are clear—the type-stamping version of the program is three times faster.

In this design, the burden of type stamping is placed on the programmer. We have experimented with alternative designs in which stamping is performed silently—either by adding a stamp whenever a sequence value is upcast to type `object` or by including a type stamp in every sequence object. However, we have not found a design in which the performance costs of stamping and stamp checking seem acceptably predictable. The difficulty is that there are infinitely many equivalent representations of the static type of a given sequence value. Because the process of stamping is invisible, the programmer has no way of predicting which of these representations will actually appear in the stamp. Thus, rather than the simple syntactic-identity check used above, we must ensure that *any* representation will produce the same effect—i.e., we must perform full equivalence checking between stamps at run time. Indeed, to avoid requiring the programmer to calculate the minimal types of sequence values (because only this type will satisfy an equivalence check), we would need to perform full subtype testing at run time. This is problematic, since—although common cases of subtype checking for regular types can be optimized sufficiently to make the compiler's front end acceptably fast in practice [16]—in general subtype testing can take time exponential in the size of the types. Such a potentially costly operation should not be applied automatically.

7 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 hybrid `pcdata` encoding described in Section 5 and the lazy append with lazy normalization policy described in Section 4.

We start by comparing Xtatic with the Qizx/Open [5] 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.

	`shake`
Xtatic	7,500 ms
Qizx/Open	3,200 ms

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.

We also implemented several XQuery examples from the XMark suite [31], 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$^\sharp$ 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 [3]. 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 memory 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.

In the next pair of experiments, we compare Xtatic with CDuce [1] 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.

	`split`	`addrbook`
Xtatic	950 ms	1,050 ms
CDuce	650 ms	1,300 ms

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 [24]. 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.[6]

	`recipe`	`sortedaddrbook`
Xtatic	250 ms	1,600 ms
Xact	60,000 ms	10,000 ms

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

[6]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.

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.

The last experiment compares Xtatic with a C$^\sharp$ program using DOM and the .NET XPath library, again using the `addrbook` example on the 25MB input file. The C$^\sharp$ program employs XPath to extract all the `APers` elements with `tel` children, destructively removes their `email` children, and returns the obtained result.

	`addrbook`
Xtatic	1,050 ms
DOM/Xpath	5,100 ms

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.

8 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$^\sharp$. 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 [8], another presenting the core language design, integrating the object and tree data models and establishing basic soundness results [9], and the third proposing a technique for compiling regular patterns based on *matching automata* [25].

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 [39] and XSLT [37] 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, 1] generalizes XDuce's type system with intersection and function types. The Xen language of Meijer, Schulte, and Bierman [27, 28] is a proposal to significantly modify the core design of C$^\sharp$ in order to integrate support for objects, relations, and XML (in particular, XML itself simply becomes a syntax for serialized object instances). Xact [24, 2] 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 [12] 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 [23] is a source to source compiler for an extension of Java that, from language design point of view, it 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 [4].

So far, most of the above projects have concentrated on developing basic language designs and there is relatively 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 [1], 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` (Section 5) 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 [21] describes the implementation of Version 6.1 of his XSLT processor Saxon. The processor is implemented in Java and, like to 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 [29] describes a technique similar to our eager normalization in their string processing language Poplar. Sleep and Holmström [32] propose a modification to a lazy evaluator that corresponds to our lazy normalization. Keller [22] 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 [18, 19, 20, 30]. They 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 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.

Another line of work, started by Hughes [17] continued by Wadler [38], and more recently Voigtlander [36], 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 [33] 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 [26] 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.

9 Future Work

We have written a number of small application programs in Xtatic, one of which is in regular use for generating the *Caml Weekly News* (see `http://sardes.inrialpes.fr/~aschmitt/cwn/`). In the coming months, we plan to use Xtatic for some more ambitious applications, to stress-test both the language design and the implementation.

A pragmatically important area for future exploration is the extension of pattern matching with XPath-like constructs. Some preliminary results on translating a "downward axis" fragment of XPath into regular patterns are reported in [10].

Acknowledgements

Parts of the Xtatic compiler were implemented by Eijiro Sumii and Stephen Tse. Conversations with Eijiro contributed many ideas to Xtatic, its implementation, and this paper. We also thank Haruo Hosoya, Alain Frisch, Christian Kirkegaard and Xavier Franc for discussing with us 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] 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] A. S. Christensen, C. Kirkegaard, and A. Møller. A runtime system for XML transformations in Java. In Z. Bellahsène, T. Milo, and e. a. Michael Rys, editors, *Database and XML Technologies: International XML Database Symposium (XSym)*, volume 3186 of *Lecture Notes in Computer Science*, pages 143–157. Springer, Aug. 2004.

[3] DataPower Technology, Inc. XSLTMark. `http://www.datapower.com/xml_community/xsltmark.html`, 2001.

[4] B. Emir. Extending pattern matching with regular tree expressions for XML processing in Scala. Diploma thesis, EPFL, Lausanne; `http://lamp.epfl.ch/~buraq`, 2003.

[5] X. Franc. Qizx. `http://www.xfra.net/qizxopen`, 2003.

[6] A. Frisch and L. Cardelli. Greedy regular expression matching. In *Workshop on Programming Language Technologies for XML (PLAN-X)*, Jan. 2004.

[7] A. Frisch, G. Castagna, and V. Benzaken. Semantic subtyping. In *IEEE Symposium on Logic in Computer Science (LICS)*, 2002.

[8] V. Gapeyev, M. Y. Levin, B. C. Pierce, and A. Schmitt. The Xtatic experience. Technical Report MS-CIS-04-24, University of Pennsylvania, Oct. 2004.

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

[10] V. Gapeyev and B. C. Pierce. Paths into patterns. Technical Report MS-CIS-04-25, University of Pennsylvania, Oct. 2004.

[11] J. Garrigue. Programming with polymorphic variants, Sept. 1998.

[12] M. Harren, B. M. Raghavachari, O. Shmueli, M. Burke, V. Sarkar, and R. Bordawekar. XJ: Integration of XML processing into Java. Technical Report rc23007, IBM Research, 2003.

[13] H. Hosoya and M. Murata. Boolean operations and inclusion test for attribute-element constraints. In *Eighth International Conference on Implementation and Application of Automata*, volume 2759 of *Lecture Notes in Computer Science*, pages 201–212. Springer-Verlag, 2003. Preliminary version in PLAN-X 2002.

[14] 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.

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

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

[17] J. Hughes. A novel representation of lists and its application to the function "reverse". *Information Processing Letters*, 22(3):141–144, Mar. 1986.

[18] H. Kaplan, C. Okasaki, and R. E. Tarjan. Simple confluently persistent catenable lists. *SIAM Journal on Computing*, 30(3):965–977, 2000.

[19] H. Kaplan and R. E. Tarjan. Persistent lists with catenation via recursive slow-down (preliminary version). In *ACM Symposium on Theory of Computing*, pages 93–102, 1995.

[20] H. Kaplan and R. E. Tarjan. Purely functional, real-time deques with catenation. *Journal of the ACM*, 46:577–603, 1999.

[21] M. H. Kay. Saxon: Anatomy of an xslt processor, Feb. 2001. `http://www-106.ibm.com/developerworks/library/x-xslt2/`.

[22] R. M. Keller. Divide and CONCer: Data structuring in applicative multiprocessing systems. In *Proceedings of the 1980 ACM conference on LISP and functional programming*, pages 196–202, 1980.

[23] M. Kempa and V. Linnemann. On XML objects. In *Workshop on Programming Language Technologies for XML (PLAN-X)*, 2003.

[24] C. Kirkegaard, A. Møller, and M. I. Schwartzbach. Static analysis of XML transformations in Java. *IEEE Transactions on Software Engineering*, 30(3):181–192, Mar. 2004.

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

[26] K. Marriott and H. Søndergaard. Difference-list transformation for prolog. *New Generation Computing*, 11:125–157, 1993.

[27] E. Meijer and W. Schulte. Unifying tables, objects and documents. In *Declarative Programming in the Context of OO Languages (DP-COOL)*, Sept. 2003.

[28] E. Meijer, W. Schulte, and G. Bierman. Programming with circles, triangles and rectangles. In *XML Conference and Exposition*, Dec. 2003.

[29] J. H. Morris, E. Schmidt, and P. Wadler. Experience with an applicative string processing language. In *ACM Symposium on Principles of Programming Languages (POPL), Las Vegas, Nevada*, pages 32–46, 1980.

[30] C. Okasaki. Amortization, lazy evaluation, and persistence: Lists with catenation via lazy linking. In *IEEE Symposium on Foundations of Computer Science*, pages 646–706, Oct. 1995.

[31] A. R. Schmidt, F. Waas, M. L. Kersten, M. J. Carey, I. Manolescu, and R. Busse. XMark: A benchmark for XML data management. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 974–985, Hong Kong, China, Aug. 2002. See also `http://www.xml-benchmark.org/`.

[32] M. R. Sleep and S. Holmström. A short note concerning lazy reduction rules for append. *Software Practice and Experience*, 12(11):1082–4, Nov. 1982.

[33] L. Sterling and E. Shapiro. *The Art of Prolog.* MIT Press, 1986.

[34] Sun Microsystems. The Java architecture for XML binding (JAXB). `http://java.sun.com/xml/jaxb`, 2001.

[35] N. Tabuchi, E. Sumii, and A. Yonezawa. Regular expression types for strings in a text processing language. In J. V. den Bussche and V. Vianu, editors, *Proceedings of Workshop on Types in Programming (TIP)*, pages 1–18, July 2002.

[36] J. Voigtländer. Concatenate, reverse and map vanish for free. In *ACM SIGPLAN International Conference on Functional Programming (ICFP), Pittsburgh, Pennsylvania*, pages 14–25, 2002.

[37] W3C. XSL Transformations (XSLT), 1999. `http://www.w3.org/TR/xslt`.

[38] P. Wadler. The concatenate vanishes. Note, University of Glasgow, 1987. (revised 1989).

[39] XQuery 1.0: An XML Query Language, W3C Working Draft, July 2004. `http://www.w3.org/TR/xquery/`.