Statically Typed Document Transformation: An XTATIC Experience

Vladimir Gapeyev François Garillot Benjamin C. Pierce

Technical Report MS-CIS-05-21
Department of Computer and Information Science
University of Pennsylvania

October 14, 2005

Abstract

XTATIC is a lightweight extension of C♯ with native support for statically typed XML processing. It features XML trees as built-in values, a refined type system based on *regular types* à la XDUCE, and *regular patterns* for investigating and manipulating XML. We describe our experiences using XTATIC in a real-world application: a program for transforming XMLSPEC, a format used for authoring W3C technical reports, into HTML. Our implementation closely follows an existing one written in XSLT, facilitating comparison of the two languages and analysis of the costs and benefits—both significant—of rich static typing for XML-intensive code.

1 Introduction

A profusion of recent language designs, including XDUCE [16, 17, 18], CDUCE [11, 2], XACT [23, 8], XQUERY [4, 10], XJ [15], XOBE [22], and XTATIC [14, 24, 12, 13, 25], are founded on the belief that rich static type systems based on regular tree languages can offer significant benefits for XML-intensive programming. Though attractive, this belief remains open to question on a number of counts. For example, are familiar XML processing idioms from untyped settings easy to enrich with types, or are there important idioms for which static typing is awkward or unworkable? Is it feasible to reimplement untyped applications in a statically typed language in a "bug-for-bug compatible" fashion? Does the need to please the typechecker lead to too much repetitive boilerplate or too many type annotations? Our aim is to put these questions to the test by a detailed comparison of a non-trivial application originally written in XSLT 1.0 [9] and a faithful reimplementation of the same application in XTATIC.

For this experiment, we chose a task that has also been used as a case study in the standard XSLT reference [19, 20]: translation of structured documents from a high-level document description language, XMLSPEC, into XHTML. XMLSPEC is the format used for authoring official W3C recommendations and drafts. This example is non-trivial but of manageable size: the DTD for XMLSPEC defines 102 elements and 57 type-like entities, while the XHTML DTD defines 89 elements and 65 entities; the XSLT stylesheet implementing the transformation is 770 lines long. Besides styling XMLSPEC elements as HTML, its functions include formatting BNF grammars, section numbering, setting up cross-references, and generating the table of contents. A useful effect of emulating a finished untyped application is that both costs and benefits are visible all at once, rather than arising and being dealt with incrementally, throughout the design and development process. To maximize the opportunities for comparison, our XTATIC implementation closely follows not only the behavior, but also, as far as possible, the structure of the original XSLT implementation.

The contributions of the paper are as follows. First, we draw attention to the XMLSPEC problem itself. This problem offers a good balance of size, complexity, and familiarity, and we hope that it can be re-used by others as a common benchmark for XML processing languages. Second, we present a detailed analysis of the costs and benefits of expressive static types for XML manipulation, both of which were substantial in this application. The main cost is the difficulty of inferring appropriate types for multiple, mutually recursive transformations. The main benefit is the expected one: exposure of design flaws in the XMLSPEC DTD—which, in the XSLT stylesheet, show up as behavioral bugs—as type inconsistencies instead. Third, we demonstrate that the type system and processing primitives of XTATIC are sufficiently powerful and flexible to fix (or gracefully work around) these bugs without modifying the XMLSPEC DTD. Fixing some of them in the XSLT stylesheet appears more difficult, if indeed it is possible at all. Finally, reimplementing an existing stylesheet gives us many opportunities for head-to-head comparisons of XSLT and XTATIC, highlighting areas where each shines. In particular, we observe that XTATIC-style regular pattern matching is much more natural than XSLT's style—structural recursion augmented with "context probing"—when processing structures, such as the BNF grammar descriptions found in XMLSPEC, where ordering is important. Conversely, XSLT is natural for straightforward structural traversions with local transformations, whereas XTATIC requires a heavier explicit-dispatch control flow. Also, XSLT's data model, which treats the original document as a resource for the computation, is extremely convenient for certain tasks, though we can mimic some of its uses with generic libraries in XTATIC.

Section 2 summarizes XMLSPEC and gives a high-level explanation of the transformation task. Section 3 describes the main challenges of expressing the core XSLT processing model in XTATIC. Section 4 compares the processing of structured data such as BNF grammars in XSLT and XTATIC. Section 5 describes the auxiliary data structures that our application uses in place of the global document access primitives offered by XSLT. We close with an overview of related languages in Section 6 and some concluding thoughts in Section 7. The paper is intended to be self-contained, but it does not present the motivations or technical details of the XTATIC design in depth; for these, the reader is referred to our earlier papers, especially [13, 14].

2 The Problem

The history of both XSLT and XMLSPEC goes back to 1998, when the standards for XML and XSLT themselves were still under development. Successive versions of the DTD and the stylesheet (available from the

```
<body>
<div1 id='sec-intro'> <head>Introduction</head>
<p>XML is an application profile or restricted form of SGML
<bibref ref='ISO8879'/>.</p>

<div2 id='sec-origin-goals'> <head>Origin and Goals</head>
<p>The design goals for XML are:
<olist>
<item><p>XML shall be usable over...</p></item>
<item><p>XML shall support...</p></item>
</olist></p>
</div2>
<div2 id='sec-terminology'> <head>Terminology</head>
<p>The terminology used to describe XML...</p>
</div2></div1>
</body>

<back>
<div1 id='sec-bibliography'> <head>References</head>
<blist>
<bibl id='ISO8879' key='ISO 8879'>
ISO. <emph>ISO 8879:1986(E). Standard Generalized Markup
Language (SGML).</emph> First edition 1986-10-15.
[Geneva]: ISO, 1986. </bibl>
</blist>
</div1>
</back>
```

1 Introduction

XML is an application profile or restricted form of SGML [ISO 8879].

1.1 Origin and Goals

The design goals for XML are:

1. XML shall be usable over...
2. XML shall support...

1.2 Terminology

The terminology used to describe XML...

A References

ISO 8879

ISO. *ISO 8879:1986(E). Standard Generalized Markup Language (SGML).* First edition 1986-10-15. [Geneva]: ISO, 1986.

Figure 1: A sample XMLSPEC document fragment and its rendering via HTML

XMLSPEC web page, `http://www.w3.org/2002/xmlspec/`) continue to be used for developing W3C specifications.

Our development is based on the 1998 version of XMLSPEC—the one used for the original XML Recommendation. Our primary reason for using this somewhat dated version was the public availability of the XML sources for the Recommendation, which we used as testing data. More recent W3C specifications, developed with newer versions of XMLSPEC, appear to be available only in formatted (HTML, PS, PDF) form. The XMLSPEC Guide [27] is a useful resource for understanding the XMLSPEC DTD (although it describes a later, more feature-rich version). The original 1998 XSLT stylesheet is described in detail in the 2nd edition of Michael Kay's XSLT reference [19]. Both the DTD and the stylesheet are available from the book's web page (`http://www.wrox.com`).

XMLSPEC is similar to more elaborate XML-based document schemas, such as DOCBOOK (`http://www.docbook.org/`) and the Text Encoding Initiative (`http://www.tei-c.org/`), in that it encodes the "logical" structure of a document so that the same information can be presented in different styles and media. Here, we consider only the task of transforming an XMLSPEC document into a single HTML page, an example of which appears in Figure 1.

Since both XMLSPEC and XHTML are used for document markup, there are many similarities between their DTDs. In both, a valid document file has distinct sections for meta-data and for the content proper. The content has three distinct kinds of markup: top-level, or sectional, for hierarchical document organization; medium-, or paragraph-level, for chunks of actual content; and low-, or phrase-level, for the content flow itself. More interesting for our task, though, are the differences, which stem from differences in purpose between logical and presentation markup: XMLSPEC uses markup to indicate the purpose of a piece of text in the discussion of a subject matter, while HTML uses markup to instruct a browser how a piece of text should be visually presented to the reader.

For example, the hierarchical document structure is represented explicitly in XMLSPEC by nested sectional elements `div1`, ..., `div4`, while in HTML it is implied by heading elements `h1`, ..., `h6` that interrupt the flow of paragraph-level markup. Both formats include generic paragraph-level elements—for example, enumerated and bulleted lists (`ol` and `ul` in HTML vs `olist` and `ulist` in XMLSPEC) and paragraphs (`p` in both). But XMLSPEC also defines special-purpose variants like `blist`, which is a list containing only bibliographic entries, and `vcnote`—a special kind of paragraph for technical snippets called validity constraint notes. Finally, at the phrase level, where HTML elements like `em`, `i`, `b` decorate the flow of character text with visual emphasis and the anchor element `a` provides simple linking points and links for external resources or

locations in the document, their XMLSPEC counterparts play more semantically-loaded roles. For example, a `termdef` element encloses a phrase that defines the meaning of a term (whose occurence in the definition is marked by a `term` element) and can be linked to from other parts of the document by element `termref`. There are many more elements for specific roles, such as language keywords (`kw`), references (`specref`) to other parts of the specification, etc. This semantic specialization of elements allows one to vary independently not only their visual representation, but also additional processing such as creation of indexes and glossaries.

Another category is XMLSPEC elements containing "structured data" of various kinds. The most interesting example is the `scrap` element, which encapsulates BNF rules for grammar productions; its formatting is discussed in detail in Section 4.

Most of the task of an XMLSPEC to HTML transformer is thus a straightforward (often literally one-to-one) mapping from XMLSPEC to HTML element tags. But there are several aspects that are more interesting, including displaying structured data in a readable form, computing section numbers based on the hierarchical positioning of `div` elements, creating a table of contents, with entries hyperlinked to the corresponding sections, and formatting the cross-references occurring in the document so that they properly mention features of the referent, such as title or its computed section number.

3 Structural Recursion

The processing model of XSLT is rather different from the explicit control flow of traditional programming languages, including XTATIC, being based on an *implicit* recursive traversal of the input document. After introducing the XSLT processing model and sketching how an XTATIC application can simulate it explicitly, this section discusses the main challenges of making this implementation strictly typed: (1) structuring its code to accomodate the constraints of typing and (2) fixing the typing bugs inherited from the stylesheet.

3.1 Implicit Structural Recursion in XSLT

An XSLT *stylesheet* is a collection of *templates*, each specifying a computation to be performed on document nodes that satisfy a specified test condition. The execution of a stylesheet proceeds in a single recursive pass over the input document, in document order. For each node encountered during the traversal, the run time system selects the most specific template whose test is satisfied by the node and executes it. Consider, for example, the following template:[1]

```
<xsl:template match="olist">
  <ol> <xsl:apply-templates/> </ol>
</xsl:template>
```

The test (`match="olist"`) says that the template is applicable to XMLSPEC `olist` elements; for each such element, the template produces an HTML `ol` element. The contents of the `ol` are the result of a further recursive traversal of the input: the instruction `<xsl:apply-templates>` designates the location receiving the result of applying the same procedure of selecting and executing an appropriate template, to each child node of the `olist`, in order. Since, according to the XMLSPEC DTD, the only possible children of `olist` are `item` elements, the template that gets invoked on them is

```
<xsl:template match="item">
  <li> <xsl:apply-templates/> </li>
</xsl:template>
```

which similarly constructs an HTML `li` element from each XMLSPEC `item` element. The recursive descent of the traversal terminates either on the document's text nodes, which get copied into the output, or on templates that do not call others via `<xsl:apply-templates/>` or a similar instruction.

[1] The XML-based syntax is a controversial aspect of XSLT. Readers unfamiliar with the language only need to know that elements starting with the `xsl:` prefix are XSLT instructions, while others are literal elements constructing the output.

In general, the test condition in a template's `match` attribute is specified by an *XSLT pattern*, which is written in the downward subset of XPATH. A template is applicable to an element when its pattern *matches* it, i.e., there is an ancestor node, starting from which the pattern (as a path) would select the element. More than one template can be applicable to a document node, but there is always at least one, since XSLT predefines a default template applicable to any element, whose action is to proceed with the traversal without producing any output. In the case of multiple applicable templates, only one of them gets selected for execution according to a set of priority rules whose particulars are not important for this discussion.

The bulk of the XMLSPEC stylesheet consists of templates similar to these, performing simple tag-to-tag transformations—sometimes augmented with other output whose generation depends only on the current element. This processing style, known as *structural recursion* [1, 5, 6], is the backbone of the XSLT processing model. However, since a simple one-pass structural recursion alone would not be sufficient for many applications, XSLT augments it with more features, some of which we will see later.

3.2 Types and Patterns in XTATIC

Before describing our implementation of the formatter, let us pause, briefly, to review the XML types and patterns found in XTATIC.

XTATIC's types are composed from XML element tags using the familiar regular expression operators of concatenation (","), alternation ("|"), repetition ("*"), and non-empty repetition ("+"). They can also contain type names, which are bound to their definitions by top-level `regtype` declarations. For example, here is a fragment of the XTATIC type declarations corresponding to the XMLSPEC DTD:

```
regtype s_olist      [[ <olist> s_item+ </> ]]
regtype s_item       [[ <item> s_obj_mix+  </> ]]
regtype s_obj_mix    [[ s_p | s_olist | s_ulist | ... ]]
```

We use the prefix `s_` for type names coming from XMLSPEC, and, later, `h_` for names coming from XHTML. The double square brackets are used to separate regular types, patterns, and XML values from surrounding C♯ code.

The semantics of XML types is similar to that of regular expressions on strings: a type is the set of values described by the type's definition, except that the values are XML document fragments—i.e. sequences of trees built from XML element tags and characters. For example, the values of type `s_item` are single XML elements of the form `<item>...</item>` whose contents are non-empty sequences of elements described by the union type `s_obj_mix`. The predefined type `xml` describes all well-formed XML values. The brackets with no content, `[[]]`, denote the type containing only the empty sequence (when used where a type is expected), as well as the empty sequence value itself (when used where a value is expected).

A regular pattern is a type annotated with variables. For example,

```
[[<olist> s_item first, s_item+ rest </>]]
```

is a pattern with variables `first` and `rest` that will be bound to values of types `s_item` and `s_item+` after a successful match. An XML value matches a pattern when the value belongs to the type obtained by erasing the bound variables. These patterns are the main construct that XTATIC programs use to analyze XML.

3.3 Explicit Structural Recursion in XTATIC

Implementing an *untyped* equivalent of the XMLSPEC stylesheet's behavior in XTATIC is straightforward: it can be written as a collection of mutually recursive static class methods, one per template, plus a dispatcher method that simulates the role of the XSLT run-time system. Figure 2 shows the fragments of this implementation corresponding to the two XSLT templates discussed above.

The two template methods have a similar structure. `TemplateItem`, for example, declares `s_item`, the type of XMLSPEC elements on which it can operate, as its input type; then, relying on the fact that the argument `elt` can only contain an `item` element, it uses a pattern assignment to extract the element's content into the variable `cont`; finally, it builds and returns the resulting HTML `li` element. The contents of `li` come

```
static [[xml]] TemplateOlist ([[s_olist]] elt){
  [[<olist>xml cont</>]] = elt;
  return [[<ol>Dispatch(cont)</>]];
}

static [[xml]] TemplateItem ([[s_item]] elt){
  [[<item>xml cont</>]] = elt;
  return [[<li>Dispatch(cont)</>]];
}
```

```
static [[xml]] Dispatch ([[xml]] seq) {
  [[xml]] res = [[]];
  while (!seq.Equals([[]])) {
    match (seq) {
      case [[s_olist elt, xml rest]]:
        res = [[res, TemplateOlist(elt)]];
        seq = rest;
      case [[s_item elt, xml rest]]:
        res = [[res, TemplateItem(elt)]];
        seq = rest;
      //......     }}
  return res; }
```

Figure 2: A fragment of the untyped structural recursion code in XTATIC.

from a call to the method `Dispatch`, which plays the role of the `<xsl:apply-templates>` instruction. Note that the pattern in the assignment follows the definition of the type `s_item`.

The `Dispatch` method uses a combination of C♯ `while` and XTATIC `match` statements to consume the input sequence from the variable `seq` and produce the output sequence in the variable `res`. XTATIC's `match` statement is similar to C♯'s `switch`, but its `case` tests are patterns and therefore can assign fragments of the input to variables for use in the clause's body. The full code of `Dispatch` contains a `case` for each XMLSPEC element, except for elements involved in presenting structured data, which are not covered by the dispatching framework (see Section 4).

3.4 Typing the Recursion

Our goal, however, is to implement a well-typed formatter—i.e., one whose output is, by construction, valid HTML for any valid XMLSPEC input. Therefore, we need to give more precise output types to our methods.

Almost every template method returns a sequence of one or more HTML elements that it creates itself; in these cases, the precise output type for the method can be inferred from its code alone. For example, `TemplateItem` is intended to return values of type `h_li`. Precise template method types induce a precise result type for `Dispatch`, which, instead of `xml`, now yields the union of the result types of all the templates it invokes.

This type, however, is too large. For example, in order for `TemplateOlist` to return a valid `ol` element, the static result type of the recursive call to `Dispatch` at this point must contain only `li` elements. Thus, instead of a single `Dispatch` method, we need to define several dispatchers, each invoking only the subset of template methods suitable for a particular context and therefore ensuring appropriate input and output types. For example, the typed version of `TemplateItem` becomes:

```
static [[h_li]] TemplateItem ([[s_item]] elt){
  [[<item>s_obj_mix+ cont</>]] = elt;
  return [[<li>DispatchInItem(cont)</>]];
}
```

Besides the precise return type and the call to the custom dispatcher `DispatchInItem`, it also analyzes input `elt` by a pattern that strictly follows the definition of type `s_item` and therefore gives the variable `cont` a more precise type, on which `DispatchInItem` can rely.

In general, the dispatcher used by a template must be prepared to handle any input that the template can pass to it, and its output must be acceptable for the use the template has for it. Any collection of dispatchers that satisfy these constraints for all templates would give a type-correct formatter. For a few of the templates, however, it is not possible to compose a well-typed dispatcher from the template methods that would faithfully reproduce the operation of the stylesheet's templates. These are instances of genuine processing bugs in the original XSLT application, which can only be fixed by modifying existing or writing additional template code.

In a few cases, the bugs are caused by subtle incompatibilities between XMLSPEC and HTML that are possible (though a bit tricky) to smooth out in XTATIC, but apparently not in XSLT, so it is instructive to discuss them and our solutions in some detail.

3.5 Bugs and Fixes

XMLSPEC defines an element `ednote` for recording editorial remarks. The DTD allows `ednote` to appear in both paragraph- and phrase-level contexts, but the XSLT stylesheet contains only one template for `ednote`, which formats it as `blockquote`, a paragraph-level HTML element presented in browsers as an indented paragraph. Clearly, appearances of ednote in phrase-level contexts (e.g., inside `head` elements of section titles) should be formatted differently. To handle this, we implement a second template method for `ednote`, with a phrase-level-friendly return type. A dispatcher that has the `ednote` element in its input type processes it with whichever of the two template methods that is compatible with the dispatcher's return type.

A similar, but trickier, problem arising in the formatting of another phrase-level XMLSPEC element, `quote`. This element is different from most others: rather than creating a new HTML element or two, the corresponding template just surrounds the result of recursively formatting the `quote`'s contents with quotation marks. The content type of `quote` is such that it gets transformed into output belonging to the most general HTML phrase-level type, `h_Inline`. One of the elements that can occur inside `h_Inline` is the anchor element `a`, and the content of the latter is described by the subtype `h_a_content` of `h_Inline`, which disallows `a` elements, prohibiting nested anchors. The `quote` element itself, however, can occur in an XMLSPEC context that ends up formatted inside an `a` element, possibly producing a nested anchor. The resolution in XTATIC is similar to the one for `ednote`: we write two template methods for `quote`, both just adding quotation marks, but to the results coming from two different dispatchers.

The solutions for these two problems work because, by explicitly implementing the recursive traversal as a combination of calls to several *distinct* dispatcher methods, our algorithm tracks (static) information about its current context in the input document. In principle, an XSLT stylesheet could also implement processing alternatives for `ednote` and `quote` elements, but making the context-dependent decision of which one of them to invoke would be more difficult.

The typing bug that required the most sophisticated fix in our reimplementation is caused by one of the most straightforward-looking templates in the stylesheet:

```
<xsl:template match="p">
  <p> <xsl:apply-templates/> </p>
</xsl:template>
```

This template transforms the XMLSPEC paragraph element `p` into an HTML element of the same name. The trouble is, an HTML `p` can contain only character data and phrase-level elements, while an XMLSPEC `p` can also contain select paragraph-level elements. Consequently, this template can produce an HTML `p` with paragraph-level elements, such as lists (`ol`, etc.), as children.

The sources of the XML Recommendation actually contain quite a few instances of `p` elements that tickle this bug. Since it affects validity of the generated HTML, the bug was addressed in the later versions of the stylesheet by a hack: when an element like `ol` appears inside a paragraph, the stylesheet adds to the output tree a *text* node whose content is " ", then formats the `ol`, and then generates another text node whose content is " ". This does not restore the validity of the in-memory tree produced by the stylesheet, but only of its textual serialization, implying that the stylesheet cannot be used in pipelining scenarios without re-parsing and re-validation of its output. We do not see any natural way to fix this bug in XSLT without changing the XMLSPEC DTD.

Our method `TemplateP` implements the above fix in a fully typed way. It uses a dispatcher that transforms the contents of XMLSPEC `p` into a sequence of text and phrase- and paragraph-level HTML elements, and then processes it to find (with the use of XTATIC patterns) longest subsequences of text and phrase-level elements and wrap them as HTML `p` elements. Figure 3 shows the method that performs the HTML processing pass. The final result of `TemplateP` is paragraph-level content.

From what we have said so far, it might appear that there is another way to implement `TemplateP`, not involving HTML post-processing: we could use patterns to find longest subsequences of XMLSPEC elements

```
static [[h_block*]] FlowIntoBlocks ([[h_Flow]] flow) {
  [[h_block*]] res = [[]];
  while (!flow.Equals([[]])) {
  match (flow) {
    case [[(pcchar | h_inline | h_misc_inline)+ inl, h_Flow rest]]:
      res = [[res, <p>inl</>]];
      flow = rest;
    case [[h_block+ blocks, h_Flow rest]]:
      res = [[res, blocks]];
      flow = rest;
    case [[(h_form | h_noscript) unexpected, h_Flow rest ]]:
      Error("Unexpected input in FlowIntoBlocks");
      flow = rest;
    case [[]]:Error("empty case");
  }}
  return res;
}
```

Figure 3: The method performing an HTML processing pass to detect implicit paragraphs.

and text to be transformed into phrase-level HTML, apply an appropriate dispatcher to them, and wrap the results as `p` elements. In fact, the approach we sketched above turns out to be the only one that works, because of another problem—this one caused by XMLSPEC `termdef` elements occurring in the content of `p`. These elements are used to designate boundaries of formal definitions in a specification. As with `quote`, the processing of a `termdef` does not create an HTML element—it just returns an anchor element `a` followed by the sequence resulting from processing the contents. This sequence can contain both phrase- and paragraph-level elements. If `termdef` elements only occurred surrounded by paragraph-level elements, we could implement `TemplateTermdef` like `TemplateP`. However, when an occurrence of `termdef` in `p` is directly preceded by phrase-producing content and the result produced by the `termdef` also starts with phrase-level content, the two must be joined into a single HTML paragraph. Therefore, to avoid creating spurious paragraph breaks, we define `TemplateTermdef` to just return the result of recursive processing of its contents. The latter joins the surrounding HTML and gets processed in `TemplateP` to detect the paragraphs.

Along with these significant typing difficulties, XTATIC's typechecker uncovered several more minor bugs in the stylesheet that also affected validity, but that were easy to fix by small changes to the output.

3.6 Efficiency vs. Compactness

Another issue to consider when assembling a collection of dispatchers is the tradeoff between efficiency and the compactness or clarity of the resulting code.

To obtain more efficient code, one would like to have a custom dispatcher for each distinct context where dispatchers are invoked, which contains only the templates needed to handle elements that can actually appear in this context. This approach might be called *input-driven*. At the other extreme is an *output-driven* approach, where one assembles a dispatcher for each distinct output context by collecting all the templates that produce conforming output. This approach is not always guaranteed to work (even when all template methods are well-typed) and can produce less efficient dispatchers, but can result in more compact code when the number of distinct output contexts is small. Of course, intermediate solutions are possible as long as they satisfy the typing constraints formulated earlier. In any case, additional code simplification is possible by combining commonly occurring template groups into shared dispatching subroutines.

In our application, the XMLSPEC DTD counts about 20 distinct content types, while HTML has only about a dozen, and the stylesheet exploits only about a half of those. Consequently, we tended to organize our dispatchers using the output-driven approach. At the end, our formatter has about seven distinct dispatchers which exploit subtyping relationships between their output types to arrange sharing of code, so that almost

every template gets explicitly called in the code of only one of the dispatchers. We felt that the improvement in source code size and clarity obtained in this way was more important than any likely efficiency losses.

4 Structured Data

XMLSPEC defines several collections of elements for structured data. This section uses the most sophisticated of these—elements for representing BNF grammars—as an example showing how XTATIC and XSLT handle the challenges of rendering structured data for visual presentation.

4.1 BNF Productions

A grammar fragment is represented in XMLSPEC as a sequence of production elements `prod`, each having the structure described by the following DTD declaration:

```
<!ELEMENT prod (lhs, (rhs, (com|wfc|vc)*)+)>
```

That is, a production consists of a left-hand side containing exactly one `lhs` element, which introduces the non-terminal defined by the production, and a right-hand side, which defines the unfolding of the non-terminal and consists of a sequence of one or more element groups. Each group contains exactly one `rhs` element, which represents a fragment of the unfolding (usually, an alternative BNF clause), possibly accompanied by side conditions in the form of a comment (`com`), or a reference to a well-formedness (`wfc`) or validity (`vc`) constraint. It is not important to know about the internals of the elements inside `prod`. Each of them gets formatted in the usual way as an HTML fragment to be placed inside a table cell; the layout of this table is our present concern.

[1]	lhsA	::=	rhsA	/*comA*/
[2]	lhsB	::=	rhsB1	/*wfcB1*/
			| rhsB2	
			| rhsB3	/*comB31*/
				/*comB32*/
[3]	lhsC	::=	rhsC	

Figure 4: An HTML table generated from an XMLSPEC grammar

Figure 4 shows an example. The generated table has five columns containing, respectively, an automatically generated sequence number for the production, the name of the non-terminal being defined, the symbol `::=`, the fragments of the non-terminal's definition, and the comments and constraints.

The challenge here is assigning appropriate contents to the table's cells based on the relative positioning of various elements in the flat sequence of `prod`'s children, rather than by simply reflecting a nested structure that is already present in the input.

4.2 XTATIC Solution

XTATIC's patterns address this challenge naturally. Note that, in each production, the element `lhs` contributes only to the starting of the first table row corresponding to the production, while the rest of the first row, as well as each of the remaining rows, is generated from a small "chunk" of `prod`'s children containing at most one `rhs` element and at most one `com`, `wfc` or `vc` element. This chunk can be described by the type

```
regtype xs_rhschunk [[ (s_rhs, xs_constr_mix?)  |  xs_constr_mix ]]
regtype xs_constr_mix  [[ s_com | s_wfc | s_vc ]]
```

and, using this type, can easily write patterns that split the sequence of `prod` children into the chunks necessary for creating the table row-by-row; the full code appears in Figure 5. The method `TemplateProd` starts by extracting from the production the name (`lhs`) of its non-terminal and the first `chunk` of the definition. It uses these to construct the first table row corresponding to the production in the newly created variable `res`. The number placed in the first table cell is extracted, based on the production's identifier (`prodid`), from an index data structure created before processing the document (this process is described in Section 5). The contents of `chunk` is processed by a separate method, `MkRhsChunk`, which performs a straightforward `match` on the two alternatives in the definition of `xs_rhschunk` type and invokes `TemplateRhs` and `DispatchFlow` to process the chunk's elements. The second part of `TemplateProd` is a `foreach` statement that iterates over

```
static [[h_tr+]] TemplateProd ([[s_prod]] markup) {
  [[<prod id=prodid> <lhs>pcdata lhs</>, (s_rhs, xs_constr_mix?) chunk,
                     xs_rhschunk* rest </prod>]] = markup;
  [[h_tr+]] res =
    [[ <tr valign=‘baseline‘>
         <td><a name=prodid/>, ‘[‘,prodindex.Number(prodid),‘]‘</>,
         <td>lhs</>, <td>‘::=‘</>,  MkRhsChunk(chunk)  </> ]];
  foreach ([[xs_rhschunk chunk]] in rest) {
    res = [[ res, <tr valign=‘baseline‘>
                     <td/>, <td/>, <td/>, MkRhsChunk(chunk) </tr> ]]; }
  return res;
}
static [[h_td, h_td]] MkRhsChunk ([[xs_rhschunk]] chunk) {
  match (chunk) {
    case [[ s_rhs rhs, xs_constr_mix? constrOPT ]]:
      return [[ <td>TemplateRhs(rhs)</>,  <td>DispatchFlow(constrOPT)</>]];
    case [[            xs_constr_mix constr ]]:
      return [[ <td/>,  <td>DispatchFlow(constr)</>]];  }
}
```

Figure 5: BNF production formatting in XTATIC.

the `rest` of the production by cutting consecutive chunks off it with the `[[xs_rhschunk chunk]]` pattern, while adding to `res` a new table row for each chunk.

4.3 XSLT Solution

Performing the same computation in XSLT is more difficult. We start with a high-level outline of the stylesheet's structure.

A child element of an instance of `prod` can be classified as a "starter" element if it provides data for the first non-empty cell in the HTML table's row, and as a "follow-up" element otherwise. Following this observation, the stylesheet defines two templates for each child element type of `prod`: a "cell" template that just performs formatting inside the HTML table's cell (in other words, a cell template is an ordinary structural recursion template in the sense of Section 3), and a "starter" template that is supposed to be executed only on starter elements, performing, among other things, row padding with empty cells.

Now, the order of template execution on an instance of `prod` is as follows. First, the template for `prod` detects all the starter elements among the `prod`'s children and invokes a type-appropriate starter template on each. The starter template pads the row with empty cells (or, in case of `lhs`, starts a new row, and makes cells with a running sequence number and the `::=` symbol), calls an appropriate cell template on the current element to format the its own cell, and finally formats any remaining cells in the row by applying cell templates to the appropriate following siblings of the current element.

This algorithm requires features of XSLT that go beyond structural recursion—the ability to control selection of both templates and nodes during traversal (to invoke either starter or cell templates as appropriate) and to obtain information about the surroundings of the current node.

The selection of templates to be considered for application when executing `xsl:apply-templates` can be controlled in XSLT by template *modes*. A template's definition can contain (in the start tag of `xsl:template` element) an attribute `mode` specifying the mode of this template. E.g., the "cell" templates in our stylesheet are headed by tags like

```
<xsl:template match="rhs" mode="cell">
```

Then, an `xsl:apply-templates` instruction that also mentions the `mode` attribute, e.g.

```
<xsl:apply-templates mode="cell">
```

considers only the templates marked by the same mode.

To control the selection of nodes to be processed by further traversal, the XSLT `xsl:apply-templates` instruction can be augmented with the attribute `select` specifying the sequence of nodes to be processed next, instead of the default children sequence of the current element. For example, the `prod` template restricts further processing to starter elements only by executing the instruction

```
<xsl:apply-templates
   select="child::*[self::lhs
                    or (self::rhs
                        and not(preceding-sibling::*[1][self::lhs]))
                    or ((self::vc or self::wfc or self::com)
                         and not(preceding-sibling::*[1][self::rhs]))
                   ]"/>
```

The contents of `select` is an XPATH *path expression* that, when applied to a node, produces a sequence (possibly empty) of nodes from the document that are related to the original node as specified by the path.

For our current purposes, we can think of an XPATH path as an expression of the form[2] a`::`n`[`q_1`]` ... `[`q_k`]` where a is an *axis*, n is a *node test*, and q_i are *predicates*. The execution of a path consists of taking the sequence of nodes specified by the axis a and successively pruning it to contain only the nodes satisfying both the node test n and all the predicates q_i. XPATH predefines several kinds of axes. The ones relevant to our examples are `self`, that produces the single-element sequence consisting of the current node, `child`, that gives the children of the current node, and `preceding-sibling` and `following-sibling` that give the corresponding sibling elements of the current node. The `preceding-sibling` axis produces the nodes in reverse document order, i.e. the closest sibling comes first. A node test n is either an element name (as in, e.g., `self::lhs`), which leaves the node in the result only if the node's name is the same as the test's, or a wildcard `*` (as in `child::*`), which is satisfied by any node. A predicate q can be numeric or boolean. A numeric predicate specifies a 1-based index of the node to be selected from the current sequence. E.g., the path `preceding-sibling::*[1]` selects the closest sibling preceding the current node in the document (or the empty sequence if the current node is the first child of its parent). A boolean predicate is built, using traditional boolean connectives `and`, `or` and `not`, from elementary predicates, which coincide with path expressions. When interpreted as a predicate, a path expression is false when it returns the empty sequence, and is true otherwise.

Taking these explanations into account, one can see why the above `select` expression restricts operation of `xsl:apply-templates` to elements that would start a new row in the HTML table. Technically, the path selects (by `child::*`) all children of `prod` that are (according to the following predicate) either the `lhs` element, or an `rhs` element not immediately preceded by the `lhs`, or a side condition element not immediately preceded by an `rhs`. The templates that get invoked on the elements so selected are starter templates, since they, as well as the `xsl:apply-templates` instruction, do not specify a `mode` attribute. Since `mode` is specified by cell templates corresponding to the same elements, the cell templates are only invoked by instructions at the end of starter templates, like this one in the starter template for `rhs`:

```
<td><xsl:apply-templates  mode="cell"
     select="following-sibling::*[1][self::vc or self::wfc or self::com]" />
</td>
```

More detailed explanation of BNF formatting in the stylesheet can be found in [19, 20].

4.4 Observations

The path expressions from the BNF formatting task shown above are quite complicated—expressions of such complexity rarely appear in document-oriented stylesheets and their occurrences seem to indicate processing of the islands of structured data embedded inside documents. Even though we believe that the XPATH fragment needed for handling structured data is more complicated and difficult to master than regular

[2] More precisely, the construction described here is a *step* expression s, and a general path expression p is either a step s, or an expression of the form p/s.

patterns, it can be learned. The major difficulty for someone trying to understand how BNF formatting works in the XSLT stylesheet comes from the fact that processing of a contiguous piece of data has to be distributed across several non-contiguous pieces of code, connections between which are only loosely indicated. By contrast, the ability of XTATIC's `match` statement to keep together inspection and transformation of a piece of data constituting a logical unit allowed us to write processing methods (Figure 5) whose responsibilities can be clearly specified in terms of their input-output behavior and whose code explicitly indicates dependencies as method calls.

Another small convenience available with regular patterns but not with XPATH paths is the ability to name pattern fragments and later reference them in patterns. For example, our definition of the type `xs_constr_mix` could have improved clarity of the later patterns, while no similar shortcut is available for `[self::vc or self::wfc or self::com]` in XPATH.

5 Gathering Global Information

The data model of XSLT is more complex than the one of XTATIC, supporting the notion of a *document* as a container of interconnected nodes and a corresponding assortment of basic operations that take advantage of the richer data model. Several parts of the XMLSPEC stylesheet rely on these additional XSLT features. This section explains how we handled these tasks in XTATIC, sometimes finding a generic reusable solution, other times relying on properties specific to XMLSPEC.

In XTATIC, XML values are lightweight, immutable, shareable trees, which can only be inspected in a top-down fashion. In XSLT, a value is a tree node that necessarily belongs to a single document. Given a node, one can retrieve the root of the document, explore the document in any direction from the node—including towards its ancestors and siblings—and randomly access nodes that have been marked by special ID attributes, which are specified to be globally unique within a valid document. Supporting all this structure makes run-time representations of XSLT values more heavyweight, but it also provides behind-the-scenes infrastructure for several common document-processing tasks that require information about the document as a whole. These include generation of section numbers, creating the table of contents, and formatting cross-references. An XTATIC version of the XMLSPEC formatter has to handle these tasks by explicitly computing a good deal of information that is automatically provided to a stylesheet by the XSLT run-time system.

The XMLSPEC cross-referencing elements can be classified into three groups, depending on the computational needs of their formatting: "hard-wired," "fetched," and "synthesized."

The XMLSPEC element for a *hard-wired* reference like `<termref def="dt-xml-doc"> XML documents </termref>` contains all the data that needs to appear in its HTML representation, which is `<a href = "#dt-xml-doc"> XML documents </a>`. Such references are straightforward to process both in XSLT and XTATIC.

In a *fetched* reference, data for the HTML presentation must be retrieved from the location in the input document to which the reference points. The elements `wfc` and `vc` (which appeared in Section 4) are fetched elements. For example, the element `<wfc def='NoExternalRefs'/>` points with its `def` attribute to the element

```
<wfcnote id='NoExternalRefs'>
  <head>No External Entity References</head>
  <p>Attribute values cannot contain external entity references.</p>
</wfcnote>
```

and should be formatted as an HTML anchor

```
<a href="#NoExternalRefs">No External Entity References</a>
```

whose contents are a heading fetched from the `wfcnote` element. The stylesheet obtains the heading with the XSLT instruction

```
<xsl:value-of select="id(@def)/head"/>
```

Here, `@def` is the value of the `wfc`'s `def` attribute, and `id()` is a built-in XSLT function that, given a token, returns the node of the current element that carries a so-called ID attribute with the token as its value. In this example, `id()` returns the above `wfcnote` element, and the following XPATH expression extracts the contents of its `head` child.

To replicate the functionality of `id()` in XTATIC, our formatter explicitly builds an index datastructure that maps IDs to elements. Fortunately, this indexing procedure is completely generic: our implementation is encapsulated in an `IdIndex` class that can be reused in other applications requiring ID support. Its usage consists of creating an `IdIndex` object, say `idindex`, at the beginning of processing by passing the document's root element to the `IdIndex` constructor and then using method calls like `idindex.Id(x)` to retrieve elements from the internally maintained index.

A *synthesized* reference is yet more complicated: its HTML formatting contains computed data not directly present in the source document. For example, the element `<specref ref="sec-predefined-ent"/>` uses the ID mechanism discussed above to point to the sectional element that starts as follows:

```
<div2 id='sec-predefined-ent'>
  <head>Predefined Entities</head>
```

The HTML formatting of this reference,

```
<a href="#sec-predefined-ent">[4.6 Predefined Entities]</a>
```

includes a computed section number.

The XSLT stylesheet computes the section number by fetching the above `div2` element via the `id()` function, and then invoking on it the instruction

```
<xsl:number level="multiple"
            count="inform-div1|div1|div2|div3|div4"
            format="1.1 "/>
```

(which appears to have been specially designed for this purpose!). This instruction uses the specifications in its its attributes to produce a formatted number.

To approximate the behavior of this instruction, we create another index, encapsulated by the class `NumberIndex`, that, for each sectional element in the document, maps the element's ID to the section's number. Again, our implementation is re-usable: the index's creation is parameterized by a boolean function that recognizes section-forming elements, and by an object of the `Countkeeper` class that provides an ADT for keeping track of hierarchical section numbers and formatting them as strings. These parameters roughly mimic the above three parameter attributes of `xsl:number`. We use another instance of `NumberIndex` to keep track of the sequence numbers of BNF grammar productions. The correct operation of `NumberIndex` depends on the existence of a unique identity for each sectional element—something that comes for free from the data model in the XSLT version. We use the `id` attribute, when one is provided, for this purpose. Otherwise (in XMLSPEC, `id` is optional on `div` elements), we create the identity by concatenating the words from the (mandatory) `head` element located under the `div`. In general, this does not guarantee uniqueness. However, the stylesheet uses the same trick to generate hyperlinks from the table of contents to titles in the main body, so we assume it is sufficient for the present application. With more effort, it should be possible to implement the interface of `NumberIndex` so that it mimics the `xsl:number` instruction with better fidelity, but we did not yet pursue this direction.

Besides reference formatting, computed sequence numbers stored in `NumberIndex` objects are used when creating section titles and grammar production entries while formatting the body of the document, as well as during creation of the table of contents. This differs from the stylesheet, which invokes the `xsl:number` instruction anew whenever a number needs to be generated—indeed, a naive XSLT implementation could end up repeating the same computation many times.

The table of contents itself is created by a separate document traversal, after the creation of the indexes but before formatting the document. It is implemented by three nested `foreach` loops, one for each of the three sectional levels (`div1`, `div2`, `div3`) that need to be reflected in the table of contents. This almost literally repeats the code in the stylesheet, which also uses explicit traversal (`xsl:for-each` instructions) for this task.

Each of the tasks discussed in this section (creating the table of contents and the indexes for IDs, section numbers, and production numbers) requires a pass over the document in addition to the main formatting pass. We experimented with combining some of these traversals (e.g., formatting the table of contents concurrently with the body of the specification, or creating all the indexes together), but concluded that increased complexity of the application code did not seem to us to justify the minor efficiency gains.

The experiences described in this section highlight the distinction between the general-purpose character of XTATIC and the domain-specific character of XSLT: all XSLT applications have to bear with the overhead of a heavier data model, while only those XTATIC applications that require global document information need to maintain corresponding data structures. Overall, we have been satisfied, in this application, with the ability of the general-purpose facilities of XTATIC (those it inherits from C♯) to simulate the whole-document features of XSLT, even without direct support from XTATIC's XML data model.

6 Related Work

Further details about XTATIC can be found in several earlier papers. The core language design is presented in [14], which shows how to integrate the object and tree data models and establishes basic soundness results. A technique for compiling regular patterns based on *matching automata* is described in [24] and extended to include type-based optimization in [26]. The run-time system of XTATIC is described in [12]. A critical evaluation of the main language design choices can be found in [13]. These papers, particularly [13], also offer detailed comparisons between XTATIC and a number of related language designs; we refer the interested reader to these existing discussions, rather than repeating them here.

Most of the recent crop of statically typed XML processing languages have been tested on non-trivial applications, but only rarely have these experiences been recorded in print. A notable exception is the XQUERY Use Cases[7]—a collection of small examples specifically designed to illustrate typical tasks for which XQUERY is expected to be used. Although they were created to illustrate the capabilities of particular features of XQUERY rather than to address a particular large application, they reflect the practical experience of the XQUERY editors and cover a usefully diverse set of small transformation and extraction tasks. In the absence of more practical benchmarks, the XQUERY use cases have been used to demonstrate capabilities of competing technologies, such as XSLT and CQL [3].

7 Conclusions

XSLT and XTATIC are quite different animals. XSLT is a high-level language with processing model founded in structural recursion, path-based XML manipulation primitives, and no static type system. XTATIC extends a general-purpose programming language with a more familiar imperative processing model and processes XML using regular patterns, which are tightly coupled to its static XML type system. The experience described in this paper shows that, like XSLT, XTATIC is well suited for implementing at least some document-processing applications, and that, unlike XSLT, its flexible static XML type system is capable of exposing a range of design and implementation errors and facilitating fixes. We have also observed that, even in this single application, there are some practical programming tasks that are much better served by XSLT than by XTATIC and some where XTATIC is significantly better than XSLT.

Designing a strictly typed structural recursion to match the behavior of an existing untyped implementation turned out to be a surprisingly labor-intensive process. The discussion in Section 3 demonstrates that mimicking the implicit structural recursion of XSLT by explicit recursive code is possible, even while ensuring strict static typing, smoothing architectural incompatibilities of the input and output DTDs, and maintaining a clean program organization that bears a close resemblance to the original XSLT code. Unfortunately, the details of the solution described there required significant effort to discover, over multiple cycles of trial and error. A major difficulty that one faces during type debugging is finding answers to lots of questions about relationships between types from a large collection that a DTD like XMLSPEC or XHTML constitutes. We hope that reading about our experience could help programmers facing similar typing tasks to find their solutions faster. It is likely, however, that the amount and difficulty of work needed to figure out a correct solution can be intimidating and prohibitive for a typical XML-literate C♯ programmer in a typical project. If worst

comes to worst, XTATIC lets one to escape typing quandaries (and postpone discovery of typing problems till run time) by using the generic `xml` type and unsafe casts. Taking these difficulties into account, the refined typing of XTATIC might be considered overkill for one-time-use scripts, where it may be easier to just fix the bugs upon running into their manifestations. On the other hand, the benefits of early error discovery and safety guarantees of well-typed code—compared to the current mainstream technologies where only testing is available—can outweigh the development costs in projects aiming to create reusable document processing tools.

Another conclusion from our experience is that XSLT templates—especially in their simplest form, unburdened by other XSLT features like non-downward paths—are a very convenient approach to programming structural recursion. A template-like construct for implementing *local* structural recursion (i.e., traversals that can be explicitly applied to chosen document fragments as opposed to being a carrier of the whole program's computation) would be a very useful addition to XML processing languages with explicit control flow. It would be necessary, however, for this construct to be accompanied by expressive and flexible typing rules that minimally burden the programmer.

However, having discussed these difficulties with XTATIC, we should also emphasize that XSLT, for its part, turned out to be convoluted (or worse) when faced with the need to deviate from straightforward structural recursion. For the most significant typing bugs discussed in Section 3, we do not see how they could be eliminated from the XSLT stylesheet in a natural and type-safe way without revising the XMLSPEC DTD; also, processing of structured data (Section 4) is much trickier in XSLT.

It would also be interesting to see how the original stylesheet might be adapted to a statically typed variant of XSLT. Unfortunately, for the moment, this must remain a thought experiment: post-processing validation appears to be the only form of typing control available with XSLT. XSLT 1.0 [9] is untyped, and the XSLT 2.0 Draft [21] describes only a dynamic type system, leaving possible static variants to the discretion of implementations.

Acknowledgments

We are grateful to Michael Levin and Alan Schmitt, our collaborators on the XTATIC design and implementation, for numerous discussions about practical programming in XTATIC, and in particular to Alan Schmitt for comments on an earlier draft of the paper.

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] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web: From Relations to Semistructured Data and XML*. Morgan Kaufmann, 2000.

[2] 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.

[3] V. Benzaken, G. Castagna, and C. Miachon. A full pattern-based paradigm for XML query processing. In *Practical Aspects of Declarative Languages (PADL), Long Beach, CA*, volume 3350 of *LNCS*, pages 235–252. Springer, Jan. 2005.

[4] S. Boag, D. Chamberlin, M. F. Fernández, D. Florescu, J. Robie, and J. Siméon. XQuery 1.0: An XML Query Language. Working draft, W3C, Sept. 2005. `http://www.w3.org/TR/xquery/`.

[5] P. Buneman, M. Fernandez, and D. Suciu. UnQL: A Query Language and Algebra for Semistructured Data Based on Structural Recursion. *VLDB Journal*, 9(1):76–110, 2000.

[6] P. Buneman, S. Naqvi, V. Tannen, and L. Wong. Principles of programming with complex objects and collection types. *Theoretical Computer Science*, 149(1):3–48, September 1995.

[7] D. Chamberlin, P. Fankhauser, D. Florescu, M. Marchiori, and J. Robie. XML Query use cases. Working draft, W3C, Sept. 2005. `http://www.w3.org/TR/xquery-use-cases/`.

[8] 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.

[9] J. Clark. XSL Transformations (XSLT) Version 1.0. Recommendation, W3C, Nov. 1999. `http://www.w3.org/TR/xslt`.

[10] D. Draper, P. Fankhauser, M. F. Fernández, A. Malhotra, K. Rose, M. Rys, J. Siméon, and P. Wadler. XQuery 1.0 and XPath 2.0 Formal Semantics. Working draft, W3C, Sept. 2005. `http://www.w3.org/TR/xquery-semantics/`.

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

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

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

[14] 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.

[15] M. Harren, M. Raghavachari, O. Shmueli, M. G. Burke, R. Bordawekar, I. Pechtchanski, and V. Sarkar. XJ: facilitating XML processing in Java. In *International World Wide Web Conference*, pages 278–287, 2005.

[16] 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.

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

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

[19] M. Kay. *XSLT Programmer's Reference*. Wrox, 2nd edition, 2003.

[20] M. Kay. *XSLT 2.0 Programmer's Reference*. Wrox, 3rd edition, 2004.

[21] M. Kay. XSL Transformations (XSLT) Version 2.0. Working draft, W3C, Sept. 2005. `http://www.w3.org/TR/xslt20`.

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

[23] 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.

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

[25] M. Y. Levin. *Run, Xtatic, Run: Efficient Implementation of an Object-Oriented Language with Regular Pattern Matching*. PhD thesis, University of Pennsylvania, 2005.

[26] M. Y. Levin and B. C. Pierce. Typed-based optimization for regular patterns. In *First International Workshop on High Performance XML Processing*, 2004.

[27] E. Maler. Guide to the W3C XML specification ("XMLspec") DTD, version 2.1. Technical report, W3C Consortium, 1998. `http://www.w3.org/XML/1998/06/xmlspec-report-v21.htm`.