StaXML: Static Typing of XML Document Fragments for Imperative Web Scripting Languages *

Adam D. Bradley, Assaf J. Kfoury, and Azer Bestavros
{artdodge,kfoury,best}@cs.bu.edu

BUCS-TR-2004-xxx

Abstract

We present a type system, STAXML, which employs the *stacked type syntax* to represent essential aspects of the potential roles of XML fragments to the structure of complete XML documents. The simplest application of this system is to enforce well-formedness upon the construction of XML documents without requiring the use of templates or balanced "gap plugging" operators; this allows it to be applied to programs written according to common imperative web scripting idioms, particularly the "echo"ing of unbalanced XML fragments to an output buffer. The system can be extended to verify particular XML applications such as XHTML and identifying individual XML tags constructed from their lexical components. We also present STAXML for PHP, a prototype precompiler for the PHP4 scripting language which infers StaXML types for expressions without assistance from the programmer.

1 Introduction

The XML textual media type [15] has garnered tremendous attention from both the technical press and the scientific community. XML textually encodes structured data; XML *applications* have been formulated which reflect a number of useful types of data, from traditional Internet objects (Web page markup, email messages) to programming languages [9] to privacy contracts (P3P).

XML is easily parsed and manipulated textual encoding of tree-structured data. A well-formed XML document consists of an interleaving of *text* and *markup*; markup is structural meta-data encoded as *tags*, strings which *mark* the beginning or end of embedded substructures. To ensure the tree structure is unambiguous, tags must be balanced (each start-tag must be followed by an end-tag) and nested (an end-tag must correspond with the last unmatched start-tag preceding it). Thus, an XML parser can always reconstruct the data structure encoded by an XML document, even if it does not understand the particular semantics of that structure.

It is important to ensure that programs which are intended to produce correct XML indeed do so. Some projects have done this using domain-specific languages which transform valid XML into valid XML [9, 5]. For imperative languages, there have been two generations of techniques for static verification: The first used special *template* data types [8] which are populated in very constrained ways with non-markup text; the second employed a more lenient system in which typed *gaps* can be *plugged* using fragments of correctly balanced-and-nested XML (which may themselves contain additional gaps) [10, 2]. Work in functional languages has generally centered around manipulating an underlying typed tree data model which is then serialized as XML by processes beyond the reach of the programmer [16, 12, 13]; some work in imperative languages has also followed this model [6].

The template approach restricts program output to the point of disallowing useful structures like recursive nesting with dynamic depth. While the gap-plugging approach is far more lenient, it also imposes upon the programmer a strong parallelism between the structure of program's codepaths and the tree structure of the XML being produced; in effect, a subtree must be delineated at a single point in the program, not with separate "begin" and "end" points. In practice, this means the programmer is forced to adopt a highly functional programming style, in which she is in effect grafting together whole trees rather than progressively writing the serial content of a document (which is the normal and natural imperative web scripting style).

This paper proposes the STAXML family of type systems which employ the *stacked type syntax* to broaden the range of XML-constructing imperative programs which can be statically type-checked to include programs written in this idiom. Rather than using specialized gap plugging operators, our approach uses only typed tokenized strings and a special polymorphically typed form of concatenation which tracks and reconciles structural imbalances within XML fragment strings.

This type system is a component of ιBENCH (the Internet Programming Workbench), a broad research initiative in our department seeking to integrate useful results from the programming language and internetwork-

*This research was supported in part by the NSF (awards ANI-9986397, ANI-0095988, CCR-9988529, ITR-0113193, ANI-0205294, and EIA-0202067).

```
if ($story) {
    header($story);
    slashDisplay($story, $next, $prev);
    $d = $db->getDiscussion($story->sid);
    printComments($d);
} else {
    $message = story_not_found();
    header($message);
    print $message;
}
footer();
```

Figure 1: Distilled fragment of `article.pl` from the Slash-2.2.6 source distribution

ing communities. The *i*BENCH Initiative is particularly focused upon the elevation of useful network constructs to first-class programming language citizens and the development of accompanying type systems to enforce correctness and safety of operations upon and compositions of those constructs.

This paper is organized as follows: Section 2 presents the basics of the stacked type syntax. In Section 3, we present a simplified form of our system which illustrates its premise for a simple XML-like markup language. Section 4 presents the complete STAXML system; Section 5 then outlines extensions which allow the system to recognize particular XML applications (such as XHTML) and to more aggressively identify XML tags constructed from their lexical components. Section 6 sketches an implementation of the STAXML system in a pre-compiler for the PHP4 scripting language, and Section 7 offers concluding remarks and future directions for this work.

2 Stacked Type Syntax

The stacked type syntax is a notationally elegant way to structure an infinite type space: rather than representing the type of an expression with a single symbol (*e.g.*, int or bool), the stacked type syntax expresses the type of an expression with a stack of such symbols. Conceptually, the element at the head of the stack usually describes the most immediate type information needed in order to interpret or operate upon the value of the expression, while elements further down the stack represent latent information about what can be done with the value once the elements above them have been properly interpreted. This is illustrated by Figure 2; the value of type τ_2 can only be used if the value of type τ_1 can be properly interpreted so as to "reach" it, and so on.

The stacked type syntax is a straightforward extension to most common syntaxes. Consider this toy example of

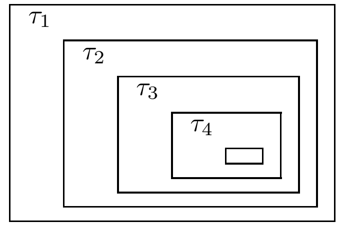

Figure 2: Conceptual Diagram of $[\tau_1[\tau_2[\tau_3[\tau_4[]]]]]$

document	::=	Chardata? (element Chardata?)*
Chardata	::=	Text containing no XML tags (STag, ETag, EmptyTag) or partial tags (unmatched or nested "<" or ">" characters); Includes comments, processing instructions, and unparsed data blocks (CDSects).
element	::=	STag *document* ETag | EmptyTag
STag	::=	<a>, , ..., <ab>, ..., and their parameterized forms (*e.g.*:)
EmptyTag	::=	<a/>, , ..., and their parameterized forms
ETag	::=	, , ...
Bad	::=	Text containing unmatched or nested "<" or ">" characters.

Figure 3: Grammar for STAXML$_S$'s Pseudo-XML

a type syntax supporting higher-order functional types:

$$
\begin{array}{rcl}
\tau & ::= & b \mid \tau \rightarrow \tau \\
b & ::= & v \mid \sigma \\
v & ::= & \mathit{int} \mid \mathit{bool} \mid \mathit{real} \mid \mathit{string} \mid \ldots \\
\sigma & ::= & [] \mid [s\ \sigma] \\
s & ::= & \mathit{stackel1} \mid \mathit{stackel2} \mid \ldots
\end{array}
$$

Intuitively, a stacked type (σ) is a LIFO list of stackable type elements (s), terminated by the empty-stack symbol ($[]$). There is no reason in principle why elements in the stack can't be drawn from the set of flat primitive types (v).

The particular semantics of *stackel1*, *stackel2*, *etc*, and of the stacking itself, are defined by the primitive constants and operators which can produce them; the syntax is generic with respect to semantics. STAXML is one particular type system with corresponding semantics which is built using the stacked type syntax; other type systems use the syntax in distinct ways [3].

$$\begin{array}{rcl} x \in \text{XmlTag} & ::= & \texttt{a} \mid \texttt{b} \mid \ldots \mid \texttt{aa} \mid \texttt{ab} \mid \ldots \mid \\ & & \texttt{ba} \mid \texttt{bb} \mid \ldots \mid \texttt{aaa} \mid \ldots \\ v \in \text{StackEntries} & ::= & \textit{STag-}x \mid \textit{ETag-}x \mid \\ & & \textit{EmptyTag-}x \mid \textit{Chardata} \mid \\ & & \textit{generalstring} \\ \sigma \in \text{Stacks} & ::= & [\,] \mid [v\ \sigma] \end{array}$$

Figure 4: Syntax of StaXML$_S$ Types

3 Simplified StaXML: StaXML$_S$

This section examines a technical precursor to the full StaXML system called StaXML$_S$. This simplified system allows strings to be typed by their conformance to a simplified Pseudo-XML language; it illustrates the premises of StaXML and its use of the stacked type syntax in an intuitively clear way not clouded by the finer details of the XML syntax which must be captured by more precise XML type systems.

3.1 Pseudo-XML

The StaXML$_S$ system tries to determine whether string expressions conform to the grammar in Figure 3. This grammar describes the same language as the *content* production in the full XML specification [4].

Note that the well-behaved members of the grammar (Chardata, STag, EmptyTag, ETag) and the set of all concatenations thereof do not exhaust the set of all strings. We therefore include the Bad terminal, which describes strings which do not correspond to any terminal or token in our grammar. Whether the concatenation of such strings may constitute a valid XML token (*e.g.,* concatenating “`<a `” with “`href="file">`” produces an STag) is beyond the power of StaXML$_S$ (see Section 5.2).

3.2 Syntax and Semantics of StaXML$_S$

Any string can be unambiguously tokenized using a longest-match lexer and the Chardata, STag, EmptyTag, ETag, and Bad descriptions from Figure 3. Each token is assigned a type value which is a valid stack element; Chardata tokens have type *chardata*, STags with the tag name “x” have types *STag*-x, and likewise ETags and EmptyTags. The syntax and semantics of StaXML$_S$ types are given in Figures 4 and 5, respectively.

A string is simply a sequence of zero or more typed tokens t_i, each with type v_i:

$$t_1 : v_1,\ t_2 : v_2,\ \ldots,\ t_n : v_n\ .$$

Every such string is characterized by a single StaXML$_S$ type (σ) which is a stack of such token types (v_is) representing the unbalanced structure of the string.

XmlTag: These correspond with the names of XML tags; *e.g.,* the name of the “`<body>`” tag is body.

StackEntries: These correspond with the simple XML-oriented tokenization of strings described above:

- ***STag*-x:** For any XmlTag x, *STag*-x represents the set of strings of the form “<x`(...)`>”, where “(...)” can be any set of whitespace-separated parameters.
- ***ETag*-x:** For any XmlTag x, *ETag*-x represents the string “</x>” and its whitespace variants.
- ***EmptyTag*-x:** For any XmlTag x, *EmptyTag*-x represents the set of strings of the form “<x`(...)`/>”, where “(...)” is as above.
- ***Chardata*:** The set of all strings which do not include tags or potential partial tags, as described for Chardata in Figure 3.
- ***generalstring*:** The set of strings including potential incomplete tags, as described by the Bad production in Figure 3. We will also use this entry to denote strings which cannot be a part of a well-formed Pseudo-XML document.

Stacks: These represent strings of tokens.

- **[]:** The set of strings described by the document production in Figure 3; strings in which all tags are properly balanced and nested. This includes strings the empty string.
- **$[v\ \sigma]$:** The set of all strings consisting of the concatenation of a string belonging to the set described by type σ, a balanced fragment (type []), a single token of type v, and another balanced fragment (type []).

Figure 5: Semantics of the Types in Figure 4

Two examples of two strings (bracketed to the left) and their StaXML$_S$ types are presented in Figure 6. For the upper string, the type [*STag*-$head$ [*STag*-$html$ []]] indicates that it consists of an “`<html>`” tag and a “`<head>`” with balanced XML fragments before, between, and after them (which are empty, empty, and “`<title>text</title>`”, respectively); in the lower string, the type [*STag*-p [*STag*-$body$ []]] functions similarly, describing a “`<body>`” and “`<p>`” with surrounding and interposed balanced fragments (empty, empty, and “`<img />`”, respectively).

3.3 The *concat*$_\mathbf{T}$ operator

For shorthand, we define a supertype, *token*, as all *STag*-x, *ETag*-x, *EmptyTag*-x, *Chardata*, and *generalstring*, and the type *string* as equivalent to the union of all Stacks types.

The type of a whole string is found by building it from its constituent tokens using a typed *token-wise concate-*

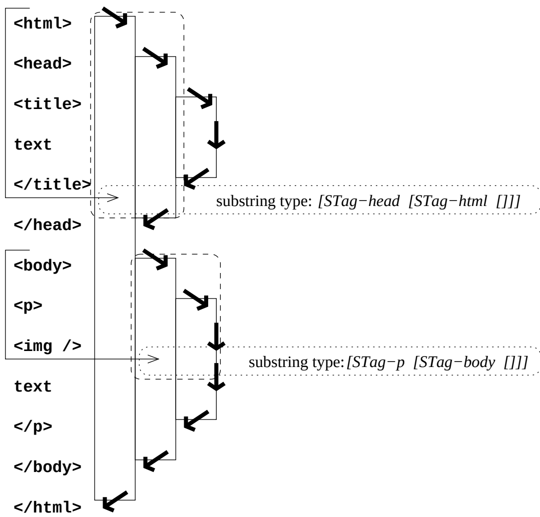

Figure 6: Strings and Their $\text{S}\textsc{ta}\text{XML}_S$ Types

nation operator, $concat_\mathrm{T}$. This operator takes two typed arguments: a *string* (a list of zero or more tokens, whose type is a Stack) and a single *token* (whose type is a StackEntry), and returns a typed string in which the token is appended to the previous list (and whose type is another Stack). $concat_\mathrm{T}$'s type can be stated most generally as:

$$\textit{string} \times \textit{token} \rightarrow \textit{string}$$

The $\text{S}\textsc{ta}\text{XML}_S$ system overloads the return type of $concat_\mathrm{T}$ to provide a much more precise type for the returned value which depends upon the precise types of the two operands. The $concat_\mathrm{T}$ operator's type is defined by a *function on types*, $typeof_{\mathrm{concat}_T}$, which has the type:

$$\text{Stacks} \times \text{StackEntries} \rightarrow \text{Stacks}$$

In other words, the type of the $concat_\mathrm{T}$ function is the result of the $typeof_{\mathrm{concat}_T}$ function as follows:

$$concat_\mathrm{T}(s : \sigma, t : v) : typeof_{\mathrm{concat}_T}(\sigma, v)$$

We define $typeof_{\mathrm{concat}_T}$ using the eight mutually exclusive and exhaustive argument pairs in Figure 7. Intuitively, the value of $typeof_{\mathrm{concat}_T}$ is the result of pushing the second argument (a StackEntry) onto the head of the first argument (a stack of StackEntries) and applying *reductions* (in the bottom-up parsing sense) which correspond to the *document* and *element* productions stated in Figure 3. In essence, the $typeof_{\mathrm{concat}_T}$ function simulates an iteration of an LL(0) bottom-up parser, in which a single symbol is examined and either shifted onto the stack (*e.g.*, Equation 3) or reduced (*e.g.*, Equation 1).

While the absence of conflicts is obvious, that our $typeof_{\mathrm{concat}_T}$ definition is also exhaustive of the input types (Stacks and StackEntries) may not be immediately clear. There are members of Stacks for which no value

$$\begin{aligned}
typeof_{\mathrm{concat}_T}(\sigma, \mathit{Chardata}) &= \sigma && (1)\\
typeof_{\mathrm{concat}_T}(\sigma, \mathit{EmptyTag}\text{-}x) &= \sigma && (2)\\
typeof_{\mathrm{concat}_T}(\sigma, \mathit{STag}\text{-}x) &= [\mathit{STag}\text{-}x\ \sigma] && (3)\\
typeof_{\mathrm{concat}_T}([\mathit{STag}\text{-}x\ \sigma], \mathit{ETag}\text{-}x) &= \sigma && (4)\\
typeof_{\mathrm{concat}_T}([], \mathit{ETag}\text{-}x) &= [\mathit{ETag}\text{-}x\ []] && (5)\\
typeof_{\mathrm{concat}_T}([\mathit{ETag}\text{-}x\ \sigma], \mathit{ETag}\text{-}y) &= [\mathit{ETag}\text{-}y\ [\mathit{ETag}\text{-}x\ \sigma]] && (6)\\
typeof_{\mathrm{concat}_T}([\mathit{STag}\text{-}x\ \sigma], \mathit{ETag}\text{-}y) &= \mathcal{E} \text{ where } x \neq y && (7)\\
typeof_{\mathrm{concat}_T}(\sigma, \mathit{generalstring}) &= \mathcal{E} && (8)
\end{aligned}$$

for all $\sigma \in$ Stacks and all $x, y \in$ XmlTag.

Figure 7: Definition of $typeof_{\mathrm{concat}_T}$ function

is defined for $typeof_{\mathrm{concat}_T}$; however, all such types are values which will never be produced by the $typeof_{\mathrm{concat}_T}$ function, and therefore will not arise in the $\text{S}\textsc{ta}\text{XML}_S$ system. Because only rules 3, 5 and 6 ever shift elements onto stacks, a tighter statement of the values of σ for which $typeof_{\mathrm{concat}_T}$ must be defined is:

$$\begin{aligned}
\sigma \in \text{AppearingSTs} &::= \mathit{startstack} && (9)\\
\mathit{startstack} &::= [\mathit{STag}\text{-}x\ \mathit{startstack}] \mid \mathit{endstack}\\
\mathit{endstack} &::= [\mathit{ETag}\text{-}x\ \mathit{endstack}] \mid []
\end{aligned}$$

Clearly, the definition of $typeof_{\mathrm{concat}_T}$ in Figure 7 exhausts this set.

3.3.1 Type Errors

Rules 7 and 8 give rise to type errors ($\mathcal{E}$), which are analogous to parsing errors for an XML parser trying to consume strings described by those type stacks; these cases represent concatenations whose results cannot safely be used as part of any well-formed XML document.

When building a practical $\text{S}\textsc{ta}\text{XML}$ compiler there exists no reason such events must correspond with compile errors *per se*; these are "errors" in the composition of strings which may be intended to constitute XML or may not; true ("hard") type errors arise only if the programmer attempts to use data of this type in a context for which it is unacceptable (*e.g.*, a function accepting only strings of type []). In common practice, we anticipate that the $\mathcal{E}$ cases in $typeof_{\mathrm{concat}_T}$ will be handled as a warning with a twofold action:

1. Since the structural problem giving rise to the error is fairly self-evident from the argument pair to $typeof_{\mathrm{concat}_T}$, the programmer is notified that an apparent illegal concatenation was performed and provided with the types and the location within the program where the erroneous concatenation appears.
2. The expression is assigned a type which indicates it can not be part of a well-formed XML document

(*e.g.*, *generalstring*), or alternately, the problematic sequence could simply be left in place in the stack to accomplish the same end. The latter also allows the compiler to "limp forward" and attempt to identify type errors in other expressions derived from the operation's result.

A more resourceful compiler-writer could, at least in some of the more commonly-occurring cases, infer the intended correct XML structure from the errant sequence of StackEntries and inject additional code (or at least recommend it to the user, complete with line numbers) which could remedy the imbalances. For example, this type:

$$[\textit{ETag-}p\ [\textit{STag-}i\ [\textit{STag-}p\ []]]]$$

represents a common HTML mistake where an appropriate corrective measure (add an "`</i>`" before the "`</p>`") is obvious (although the proper placement within the text between "`<i>`" and `</p>`" is not; hence, such a fix-up must not be done silently). Similarly, HTML programmers making the transition to XHTML often do not properly close singleton tags like "`<img/>`" or "`<hr/>`"; if the compiler knows the particular DTD or schema which the document is expected to conform to, such cases can also be easily identified and mechanically corrected.

3.3.2 Correctness of *concat*$_\mathrm{T}$

It is the definition of types with the form $[v\ \sigma]$ that supports the stack-reducing rules (1, 2, and 4) giving our type system sufficient expressive power to compactly capture XML-like well-formedness while remaining compact and computationally efficient; the system simply does not need to represent balanced subfragments explicitly in the type of the expression because they are fully represented by the implicit [] types within the stack.

Theorem 3.1. *In the* STAXML$_\mathrm{S}$ *type system, expressions of type* [] *constructed using* concat$_T$ *would be recognized by a parser as matching the "document" production in Figure 3.*

Proof. We will refer to the language described by a production using that production's name.

We prove the parallelism between such a parser and the result of our type system by induction on the structure of concatenation. The empty string by definition has type [] and conforms trivially to the document production. Now consider each of the cases in figure 7. Assuming an initial empty string s with type [] (conforms to document) and a token argument r of the appropriate type:

(1). *Chardata* is absorbed by either the leading CharData? or the trailing CharData? of the document production; this holds because a single `CharData` can capture an arbitrarily long sequence of *Chardata* tokens. Therefore:

$$\textit{concat}_\mathrm{T}\ (s : [],\ r : \textit{Chardata}) \in \text{document}.$$

(2). *EmptyTag-*x is absorbed by document as an element (one form of element is EmptyTag). Therefore:

$$\textit{concat}_\mathrm{T}\ (s : [],\ (r : \textit{EmptyTag-}x) \in \text{document}.$$

(3, 4). *STag-*x and *ETag-*x tokens are absorbed by STag and ETag, respectively, within element; this requires that the concatenation of all intermediate tokens match document, *i.e.*, have the type [], which is the case. Once an expression's type has become a non-empty stack via the application of 3, no concatenation other than 4 can return the type to []. Therefore:

$$\begin{aligned}&\textit{concat}_\mathrm{T}(\\&\quad \textit{concat}_\mathrm{T}(\\&\qquad \textit{concat}_\mathrm{T}(s : [], r : \textit{STag-}x),\\&\qquad s' : []),\\&\quad r' : \textit{ETag-}x)\ \in \text{document}.\end{aligned}$$

(5, 6). Neither of these signatures can ever give rise to a [] type, or to a type which is usable by subsequent *concat*$_\mathrm{T}$s, to produce a result type of [], so these cases are irrelevant to the proof.

(7, 8). Neither of these signatures can ever give rise to a [] type, or to a type which is usable by subsequent *concat*$_\mathrm{T}$s, to produce a result type of [], so these cases are irrelevant to the proof.

Whereas the empty string has type [] and conforms to document, and all concatenations which can give rise to strings of type [] correspond with strings also conforming to document, all expressions of type [] conform to document. □

The converse is not necessarily true because the grammar (Figure 3) does not force STag and ETag to have matching names, where STAXML$_S$ does. Thus, some strings conform to the document production but are not well-formed XML and thus have the type $\mathcal{E}$.

3.4 Generalized Concatenation

A language with only the *concat*$_\mathrm{T}$ operator to join strings is cumbersome to work with in practice. Consider concatenating the string "`<html><body>`" with a second string "`</body></html>`". The *concat*$_\mathrm{T}$ operator allows us to construct each of these two strings individually, having the types

$$[\textit{STag-}body\ [\textit{STag-}html\ []]]$$

$$\sigma\ [\textit{ETag-}x\ [\textit{STag-}x\ \sigma']] \quad \Rightarrow \quad \sigma\ \sigma' \tag{12}$$
$$\sigma\ [\textit{ETag-}y\ [\textit{STag-}x\ \sigma']] \quad \Rightarrow \quad \mathcal{E} \quad \text{where } x \neq y \tag{13}$$
$$\sigma\ [\textit{EmptyTag-}x\ \sigma'] \quad \Rightarrow \quad \sigma\ \sigma' \tag{14}$$
$$\sigma\ [\textit{Chardata}\ \sigma'] \quad \Rightarrow \quad \sigma\ \sigma' \tag{15}$$
$$\sigma\ [\textit{generalstring}\ \sigma'] \quad \Rightarrow \quad \mathcal{E} \tag{16}$$

for all $x, y \in$ XmlTag and all $\sigma, \sigma' \in$ Stacks

Figure 8: Definition of *reduce* for STAXML

and

$$[\textit{ETag-}html\ [\textit{ETag-}body\ []]],$$

respectively, but will not allow us to then concatenate the two already-assembled strings.

We therefore define the general (less restrictively typed) concatenation operator, *concat*, which operates upon two typed strings as follows. In general terms, the type of the operator *concat* is:

$$\textit{string} \times \textit{string} \rightarrow \textit{string}$$

The STAXML$_S$ type of the resulting *string* is computed by the function *typeof*$_{\text{concat}}$ which takes as arguments two members of Stacks and returns a member of Stacked Types:

$$\textit{concat}(s : \sigma, s' : \sigma') : \textit{typeof}_{\text{concat}}(\sigma, \sigma')$$

where the function *typeof*$_{\text{concat}}$ is defined as follows:

$$\textit{typeof}_{\text{concat}}(\sigma, \sigma') = \textit{fix}(\textit{reduce}, \sigma'\sigma) \tag{10}$$

and $\sigma'\sigma$ is defined as:

$$\sigma'\sigma = \begin{cases} \sigma' & \text{if } \sigma = [] \\ \sigma & \text{if } \sigma \neq [] \text{ and } \sigma' = [] \\ [v\ \sigma''\ \sigma] & \text{if } \sigma \neq [] \text{ and } \sigma' = [v\ \sigma''] \end{cases} \tag{11}$$

The function *reduce* is a stack rewriting function which searches the stack for innermost sequences of tokens corresponding with Pseudo-XML productions (Figure 3) and reduces (rewrites) them accordingly. Taking the fixed-point of *reduce* ensures that all such patterns are removed from the type of a *concat* result. This effectively allows an arbitrary number of paired tags between σ and σ' to be resolved by a single operation.

We will state the function *reduce* in terms of *type normalizations*, rules which rewrite a stack into a *normal form*. Normal forms for STAXML$_S$ types are precisely the subset of Stacks which can be constructed using the *typeof*$_{\text{concat}_T}$ operator (stated in Equation 9). The *reduce* function is defined by Figure 8.

Normalization with rules 12, 14, and 15 is confluent per Newman's Lemma [1] because the rewrite system will terminate (all rules are length-decreasing) and each of the rules is locally confluent (the rules are non-interfering). Because normalization is terminating, the result is canonical; for any given σ_1 and σ_2, there is exactly one value *typeof*$_{\text{concat}}(\sigma_1, \sigma_2)$, *i.e.*, *concat*$(s_1 : \sigma_1, s_2 : \sigma : 2)$ has exactly one STAXML$_S$ type. Normalization is performed immediately whenever inferring a STAXML$_S$ type; only normal forms are ever used as the types of expressions.

Rules 13 and 16 identify "errors" in the same sense as in *concat*$_{\text{T}}$; the compiler can terminate, or can reduce the offending patterns to *generalstring*, or can notify the user and simply leave the offending pattern in place in the stack, or could even (in some limited cases) attempt to "repair" the structure of the stack by inserting appropriately-placed additional string concatenations into the program.

Agreement of *concat* with *concat*$_{\text{T}}$ The two concatenation operators, *concat*$_{\text{T}}$ and *concat*, both recognize the same sets of strings as having the same STAXML$_S$ types, *i.e.*, any string s constructed token-wise using *concat*$_{\text{T}}$ will be judged to have the same type as if it had been constructed from other previously-concatenated strings using *concat*.

Theorem 3.2. *Consider strings $s_1 : \sigma_1$ and $s_2 : \sigma_2$, where s_2 is a list of typed tokens*

$$t_1 : v_1, \ldots, t_n : v_n$$

and σ_2 is

$$\text{typeof}_{\textit{concat}_{\text{T}}}(\cdots(\text{typeof}_{\textit{concat}_{\text{T}}}(\sigma_1, v_1), v_2) \cdots, v_n)\ .$$

Token-wise concatenation (using concat$_T$*) and general concatenation (using* concat*) of the two strings produce identical results with identical types,* i.e.*,*

$$\text{concat}(s_1, s_2) = \\ \text{concat}_T(\cdots \text{concat}_T(\text{concat}_T(s_1, t_1), t_2), \cdots t_n)$$

and

$$\text{typeof}_{\textit{concat}}(\sigma_1, \sigma_2) = \\ \text{typeof}_{\textit{concat}_{\text{T}}}(\cdots \\ \text{typeof}_{\textit{concat}_{\text{T}}}(\\ \text{typeof}_{\textit{concat}_{\text{T}}}(\sigma_1, v_1), v_2) \cdots, v_n)\ .$$

Proof. That the values are the same follows trivially from the definitions of the two operators.

In effect, both *typeof*$_{\text{concat}_T}$ and *typeof*$_{\text{concat}}$ act as rewrite systems upon the type value

```
document::= prolog element miscs
prolog::=
    XMLDecl miscs DocTypeDecl miscs |
    XMLDecl miscs |
    miscs DocTypeDecl miscs |
    miscs
miscs: misc miscs | /* nil */
misc: Comment | PI | S
element:
    EmptyTag |
    STag content ETag
content:
    Chardata morecontent | morecontent
morecontent:
    fillcontent Chardata morecontent |
    fillcontent morecontent | /* nil */
fillcontent: element | CDSect | misc
```

Figure 9: Conflict-Free BNF for the Core of XML

$[v_n\ [\ \cdots\ [v_1\ \sigma_1]\cdots]]$; this can be shown from the structure of the two functions and the correspondence between their constituent rules: 1 with 15, 2 with 14, 3 with 11, 4 with 12, 5 with 11, 6 with 11, 7 with 13, and 8 with 16.

The only difference between the two as rewrite systems is that $\mathit{typeof}_{\mathrm{concat}_T}$ prescribes a particular strategy for the order in which rewrites are performed (first try to rewrite $[v_1\ \sigma_1]$, then prepend the result thereof with v_2 and try again, *etc*) whereas $\mathit{typeof}_{\mathrm{concat}}$ does not. The two systems are therefore equivalent if all of the rewrites described are invariant to the order in which they are applied. This follows directly from their mutual non-interference, which is evident in the structure of the rule subjects. □

4 Enforcing Full XML Well-Formedness

The STAXML system we present in this section enforces many more of the basic grammatic correctness and well-formedness requirements of the XML specification [4] by more selectively reducing elements from the stack. We will express this system entirely in terms of the general concatenation operator and type normalizations.

We begin by considering the `document` and related productions from the XML specification. Since conflict-free grammars translate more easily into normalizations, we have translated the `document` EBNF into a conflict-free BNF specification, shown in Figure 9.

Clearly, a more involved system than STAXML_S is needed to handle this language; we must ensure that the document root consists of precisely one element (the `element` in the `document` production), and we must allow for a variety of ways of constructing the document prolog. To retain sufficient information in STAXML types for these purposes, we must give more narrow meanings to many types in order to preserve useful distinctions. We begin by defining the tokens which must be recognized in static strings (precisely defined in [4]):

XMLDecl Any "`<?xml ... ?>`" string.
DocTypeDecl Any "`<!DOCTYPE ... >`" string.
misc Any combination of XML comments ("`<!-- comment -->`"), PIs ("`<?xyz (...) ?>`"), and whitespace (spaces, tabs, end-of-line characters, *etc*).
***STag*-x** Any XML start-tag with the name `x`.
***ETag*-x** Any XML end-tag with the name `x`.
***EmptyTag*-x** Any self-contained XML tag with the name `x`.
Chardata Any combination of plain text and unparsed data blocks (CDSects) of the form "`<![CDATA[` (unparsed data) `]]>`".
S Any whitespace sequence (spaces, tabs, end-of-line characters, *etc*). The lexer should give precedence to assigning strings this type over *misc*.
generalstring Any string other string (*e.g.*, partial tags, unopened/unclosed comments, *etc*).
[] The empty string.

As before, stacking represents concatenation; *e.g.*, a *misc* string concatenated with a *Chardata* string has the basic type [*Chardata* [*misc* []]]. We have discarded the STAXML_S definition of [] as "a valid block of balanced pseudo-XML", and as such, there is no longer an implicit block of balanced XML between elements of the stack. Instead, we progressively reduce stack elements and sequences of stack elements to other elements representing productions in the conflict-free grammar (Figure 9), specifically:

prolog Any valid XML prolog (the `prolog` production).
xmldec Any *XMLDecl* followed by zero or more *misc*s.
dtd Any *DocTypeDecl* preceded and followed by zero or more *misc*s.
element Any complete XML element (the `element` production) preceded and followed by zero or more *misc*s.
content Any combination of *Chardata*s, *misc*s, and *element*s.
document Any valid XML document (the `document` production).

Thus, the general syntax of STAXML types is defined as in Figure 10. The types representable in this syntax must then be normalized using the rules of Figure 11.

Perhaps counterintuitively, we never normalize any stacks to the *document* type. While such an eager normalization strategy works for left-to-right parsers (like the $\mathit{concat}_{\mathrm{T}}$ operator), in the presence of arbitrarily-ordered concatenation it does not preserve enough information for us to correctly identify late prefixing of cer-

$$
\begin{array}{rcl}
x, y \in \text{XmlTag} & ::= & \mathsf{a} \mid \mathsf{b} \mid \ldots \mid \mathsf{aa} \mid \mathsf{ab} \mid \ldots \mid \\
 & & \mathsf{ba} \mid \mathsf{bb} \mid \ldots \mid \mathsf{aaa} \mid \ldots \\
v, w \in \text{StackEntries} & ::= & \textit{XMLDecl} \mid \textit{DocTypeDecl} \mid \\
 & & \textit{misc} \mid \textit{STag-}x \mid \textit{ETag-}x \mid \\
 & & \textit{EmptyTag-}x \mid \textit{Chardata} \mid \\
 & & \textit{generalstring} \mid \textit{prolog} \mid \\
 & & \textit{xmldec} \mid \textit{dtd} \mid \textit{element} \mid \\
 & & \textit{content} \mid \textit{document} \\
\sigma \in \text{Stacks} & ::= & [] \mid [v\ \sigma]
\end{array}
$$

Figure 10: STAXML Syntax of Types

tain tokens to prologs. For example, a string of the type $[\textit{element}\ [\textit{dtd}\ []]]$ is a *document*. If we later prepend this string with an *xmldec*, it is still a *document*; however, a *document* cannot be prepended by a *xmldec* because it may already begin with one. To work around this problem, we define *document* explicitly as the union of all normalized STAXML types which describe valid *document*s, presented in Figure 12. Intuitively, an expression with any of these four STAXML types can be safely promoted to type $[\textit{document}\ []]$.

This formulation supports a great deal of generality; for example, if a string may be either a *xmldec* or a *dtd*, it can still be prepended to an *element* to produce a *document*. For this reason, we define a set of supertype relations (in addition to the *document* definition) which allow such strings to preserve a sufficiently descriptive type. These relationships are represented in Figure 13, where arrows point from supertypes to subtypes. While these relations are never applied in the course of type normalization, they may be used by the type inference algorithm when trying to reconcile multiple flows of execution which produce expressions or variables with differing types (as discussed in Section 6). Notice that this is not a strict hierarchy, but includes multiple supertypes for *misc* (and, transitively, *S*) and *element*; this is not problematic, as our reconciliation algorithm depends only upon finding least common supertypes, which is straightforward so long as the relationships follow a DAG. These inter-element subtype relations are then used to reconcile full STAXML types using an algorithm given fully in [3]. Intuitively, this algorithm examines two STAXML types starting at their tails; where the corresponding StackEntries are the same or have a non-*generalstring* common supertype, that is pushed onto the result stack ρ. Failing that, the algorithm attempts to reconcile either of the tail elements with a $[]$-typed string, deferring the other tail element until the next iteration. Failing that, the *generalstring* element is pushed onto the result stack ρ. After consuming an element from the argument types, the result type ρ is

$$
\begin{array}{rcll}
\sigma\ [\textit{XMLDecl}\ \sigma'] & \Rightarrow & \sigma\ [\textit{xmldec}\ \sigma'] & (17)\\
\sigma\ [\textit{misc}\ [\textit{xmldec}\ \sigma']] & \Rightarrow & \sigma\ [\textit{xmldec}\ \sigma'] & (18)\\
\sigma\ [\textit{S}\ [\textit{xmldec}\ \sigma']] & \Rightarrow & \sigma\ [\textit{xmldec}\ \sigma'] & (19)\\
\sigma\ [\textit{DocTypeDecl}\ \sigma'] & \Rightarrow & \sigma\ [\textit{dtd}\ \sigma'] & (20)\\
\sigma\ [\textit{misc}\ [\textit{dtd}\ \sigma']] & \Rightarrow & \sigma\ [\textit{dtd}\ \sigma'] & (21)\\
\sigma\ [\textit{S}\ [\textit{dtd}\ \sigma']] & \Rightarrow & \sigma\ [\textit{dtd}\ \sigma'] & (22)\\
\sigma\ [\textit{dtd}\ [\textit{misc}\ \sigma']] & \Rightarrow & \sigma\ [\textit{dtd}\ \sigma'] & (23)\\
\sigma\ [\textit{dtd}\ [\textit{S}\ \sigma']] & \Rightarrow & \sigma\ [\textit{dtd}\ \sigma'] & (24)\\
\sigma\ [\textit{dtd}\ [\textit{xmldec}\ \sigma']] & \Rightarrow & \sigma\ [\textit{prolog}\ \sigma'] & (25)\\
\sigma\ [\textit{misc}\ [\textit{prolog}\ \sigma']] & \Rightarrow & \sigma\ [\textit{prolog}\ \sigma'] & (26)\\
\sigma\ [\textit{S}\ [\textit{prolog}\ \sigma']] & \Rightarrow & \sigma\ [\textit{prolog}\ \sigma'] & (27)\\
\sigma\ [\textit{misc}\ [\textit{element}\ \sigma']] & \Rightarrow & \sigma\ [\textit{element}\ \sigma'] & (28)\\
\sigma\ [\textit{S}\ [\textit{element}\ \sigma']] & \Rightarrow & \sigma\ [\textit{element}\ \sigma'] & (29)\\
\sigma\ [\textit{element}\ [\textit{misc}\ \sigma']] & \Rightarrow & \sigma\ [\textit{element}\ \sigma'] & (30)\\
\sigma\ [\textit{element}\ [\textit{S}\ \sigma']] & \Rightarrow & \sigma\ [\textit{element}\ \sigma'] & (31)\\
\sigma\ [\textit{Chardata}\ \sigma'] & \Rightarrow & \sigma\ [\textit{content}\ \sigma'] & (32)\\
\sigma\ [\textit{element}\ [\textit{content}\ \sigma']] & \Rightarrow & \sigma\ [\textit{content}\ \sigma'] & (33)\\
\sigma\ [\textit{misc}\ [\textit{content}\ \sigma']] & \Rightarrow & \sigma\ [\textit{content}\ \sigma'] & (34)\\
\sigma\ [\textit{S}\ [\textit{content}\ \sigma']] & \Rightarrow & \sigma\ [\textit{content}\ \sigma'] & (35)\\
\sigma\ [\textit{content}\ [\textit{misc}\ \sigma']] & \Rightarrow & \sigma\ [\textit{content}\ \sigma'] & (36)\\
\sigma\ [\textit{content}\ [\textit{S}\ \sigma']] & \Rightarrow & \sigma\ [\textit{content}\ \sigma'] & (37)\\
\sigma\ [\textit{content}\ [\textit{element}\ \sigma']] & \Rightarrow & \sigma\ [\textit{content}\ \sigma'] & (38)\\
\sigma\ [\textit{misc}\ [\textit{misc}\ \sigma']] & \Rightarrow & \sigma\ [\textit{misc}\ \sigma'] & (39)\\
\sigma\ [\textit{S}\ [\textit{misc}\ \sigma']] & \Rightarrow & \sigma\ [\textit{misc}\ \sigma'] & (40)\\
\sigma\ [\textit{misc}\ [\textit{S}\ \sigma']] & \Rightarrow & \sigma\ [\textit{misc}\ \sigma'] & (41)\\
\sigma\ [\textit{misc}\ [\textit{document}\ \sigma']] & \Rightarrow & \sigma\ [\textit{document}\ \sigma'] & (42)\\
\sigma\ [\textit{S}\ [\textit{document}\ \sigma']] & \Rightarrow & \sigma\ [\textit{document}\ \sigma'] & (43)\\
\sigma\ [\textit{element}\ [\textit{STag-}x\ \sigma']] & \Rightarrow & \sigma\ [\textit{STag-}x\ \sigma'] & (44)\\
\sigma\ [\textit{content}\ [\textit{STag-}x\ \sigma']] & \Rightarrow & \sigma\ [\textit{STag-}x\ \sigma'] & (45)\\
\sigma\ [\textit{misc}\ [\textit{STag-}x\ \sigma']] & \Rightarrow & \sigma\ [\textit{STag-}x\ \sigma'] & (46)\\
\sigma\ [\textit{S}\ [\textit{STag-}x\ \sigma']] & \Rightarrow & \sigma\ [\textit{STag-}x\ \sigma'] & (47)\\
\sigma\ [\textit{ETag-}x\ [\textit{element}\ \sigma']] & \Rightarrow & \sigma\ [\textit{ETag-}x\ \sigma'] & (48)\\
\sigma\ [\textit{ETag-}x\ [\textit{content}\ \sigma']] & \Rightarrow & \sigma\ [\textit{ETag-}x\ \sigma'] & (49)\\
\sigma\ [\textit{ETag-}x\ [\textit{misc}\ \sigma']] & \Rightarrow & \sigma\ [\textit{ETag-}x\ \sigma'] & (50)\\
\sigma\ [\textit{ETag-}x\ [\textit{S}\ \sigma']] & \Rightarrow & \sigma\ [\textit{ETag-}x\ \sigma'] & (51)\\
\sigma\ [\textit{ETag-}x\ [\textit{STag-}x\ \sigma']] & \Rightarrow & \sigma\ [\textit{element}\ \sigma'] & (52)\\
\sigma\ [\textit{EmptyTag-}x\ \sigma'] & \Rightarrow & \sigma\ [\textit{element}\ \sigma'] & (53)\\
\sigma\ [\textit{ETag-}y\ [\textit{STag-}x\ \sigma']] & \Rightarrow & \mathcal{E} \quad (\text{where } x \neq y) & (54)\\
\sigma\ [\textit{generalstring}\ \sigma'] & \Rightarrow & \mathcal{E} & (55)
\end{array}
$$

Figure 11: Normalization of STAXML types

normalized, and the algorithm repeats until both stacks are exhausted.

5 STAXML Variants

The same techniques employed to develop the STAXML system can be used to further refine it to identify more restrictive languages and to account for more general techniques for constructing XML fragments. Below we

$$
\begin{aligned}
[\mathit{document}\ []] \ = \ & [\mathit{element}\ [\mathit{prolog}\ []]] \ \bigcup \\
& [\mathit{element}\ [\mathit{xmldec}\ []]] \ \bigcup \\
& [\mathit{element}\ [\mathit{dtd}\ []]] \ \bigcup \\
& [\mathit{element}\ []]
\end{aligned}
$$

Figure 12: The [*document* []] Type

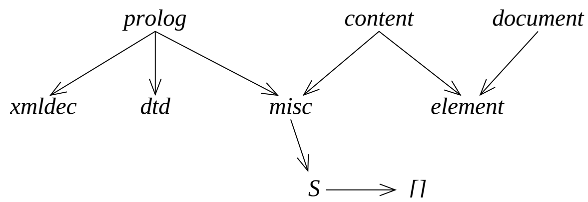

Figure 13: Sub/Supertype relationships among STAXML element types

briefly outline STAXHTML and STAXML$_A$, which reflect each of these two possibilities, respectively; complete specifications are omitted for want of space, but can be found in [3].

5.1 STAXHTML

Certain classes of XML schemata (descriptions of particular XML-based languages) can be enforced by replacing the general-purpose STAXML reduction rules presented above with specialized reduction rules which only successfully normalize the XML structures which legally fit within the schema's described XML language. As an example, we have defined a syntax of types and set of normalization rules called STAXHTML which enforces correct construction of XHTML/1.0 [14] documents.

The STAXHTML application works in principle in very much the same way STAXML does, applying very similar type normalization rules; the difference is, instead of defining rules for all x (for all possible tag names), we will define rules for particular values of x, tailoring the normalization behaviors to the particular requirements and content models corresponding with each tag name. We will also support *exclusions* (rules prohibiting particular elements from appearing as descendants of others at any depth) by augmenting each type stack element with a "taint" vector which preserves exclusion behavior without preventing otherwise valid stack-reducing normalizations.

The particular requirements of XHTML/1.0 fall into three categories:

Tags The set of allowable XML tags is a finite, defined set; no unknown attributes may appear.

$$
\begin{aligned}
\sigma\ [\mathit{element}\text{-}\mathtt{li}\ [\mathit{element}\text{-}\mathtt{li}\ \sigma']] &\Rightarrow \sigma\ [\mathit{element}\text{-}\mathtt{li}\ \sigma'] \\
\sigma\ [\mathit{ETag}\text{-}\mathtt{ol}\ [\mathit{element}\text{-}\mathtt{li}\ [\mathit{STag}\text{-}\mathtt{ol}\ \sigma']]] &\Rightarrow \sigma\ [\mathit{element}\text{-}\mathtt{ol}\ \sigma'] \\
\sigma\ [\mathit{ETag}\text{-}\mathtt{ol}\ [\mathit{STag}\text{-}\mathtt{ol}\ \sigma']] &\Rightarrow \mathcal{E}
\end{aligned}
$$

Figure 14: Type normalizations for ordinal lists

This can be easily handled by excluding from the list of normalizations rules which reduce *STag*-x, *ETag*-x, or *EmptyTag*-x for any unknown x.

Nesting For each tag name, there is specified a set of tag names which may appear as immediate children in the document tree.

This is handled by replacing the *element* type with *element*-x types for each tag name x, replacing the generic *element* consumption rules (44, 48) with tag-specific consumption rules for only the allowed child tags, and doing away with the *content* rules which subsume *element*s. Variations on this theme are needed to handle specific content models; for example, the `ol` tag must have one or more `li` children, so normalizations for *element*-li and *element*-ol are performed per Figure 14.

Exclusions Certain tags must not contain content which includes certain other tags, at any depth of nesting.

This is implemented by augmenting every stack element type with a vector of five bits, one for each of the above exclusions. All *STag*-`a`, *ETag*-`a`, and *EmptyTag*-`a` elements have the first bit set, all start and end tags of the excluded tags within "`<pre>`" have the second set, and so on. Whenever a type normalization is performed, the values of the bit field are propagated even when the particular StackEntries are rewritten to some other StackEntries; in essence, once the "taint" of an "`<a>`" tag is introduced to a type, all normalizations which may remove the explicit presence of an "`<a>`" element in the stack preserve this taint. This is complimented by the requirement that, when reducing a start-end pair corresponding to any of these exclusions, the contents must not have the excluded taint bit set; *i.e.*, when reducing a stack of the form σ [*ETag*-`button` σ' [*STag*-`button` σ'']], the type σ' must not have the third flag (corresponding with "`<button>`" exclusions) set, otherwise a type error ($\mathcal{E}$) is raised. These checks are performed in parallel with the basic normalizations derived as per the previous section.[1]

5.2 STAXML$_A$

Commonly used languages (such as PHP) will not necessarily provide us with a static tokenization/typing of text fragments which will be concatenated to form XML

```
1: echo "Class: <select name=\"class\">";
2: foreach($article_class as $ack=>$acr) {
3:   echo "<option value=\"" .
4:        htmlentities($ack) . "\"";
5:   if ($ack==$clsid)
6:     echo " selected=\"1\"";
7:   echo ">" . htmlentities($acr) .
8:        "</option>";
9: }
A: echo "</select><br />\n";
```

Figure 15: Example PHP4 Code

documents. As such, we present here an extension to our type model (based upon the concept of a finite-state lexer) for inferring the presence of such tags in modestly generalized string concatenation procedures.

Consider, as a motivating example, the snippet of PHP4 code in Figure 15, taken from an existing web application. Lines 1 and 8 include string constants which can obviously be recognized by a static string lexer as whole XML start, end, and empty tags. However, the output of lines 3-7 also represents a valid start-end tag pair; since the construction of the open tag spans multiple expressions (not to mention multiple paths of execution), a simple static string lexer is not sufficient to identify the presence of the *STag*-option substring type.

Based upon the previous sections, the reader should be able to imagine how a range of type systems could be formulated to detect such situations. In essence, all such systems will use the lexer to identify partial tags and use normalization rules to reduce concatenations thereof to the STAXML tag types discussed above. One simple approach adds three more string types to the lexer:

***OpenTag-x*:** Any string of the form “<x ” or “<x (...params...)”. (Whitespace after the x is important, as it ensures that x is the complete tag name.)

***RBracket*:** The string “>”.

***EmptyTagClose*:** The string “/>”.

and adds a few straightforward normalization rules:

$$\sigma\,[\textit{plaintext}\ [\textit{OpenTag-}x\ \sigma']] \quad\Rightarrow\quad \sigma\,[\textit{OpenTag-}x\ \sigma'] \tag{56}$$

$$\sigma\,[\textit{RBracket}\ [\textit{OpenTag-}x\ \sigma']] \quad\Rightarrow\quad \sigma\,[\textit{STag-}x\ \sigma'] \tag{57}$$

$$\sigma\,[\textit{EmptyTagClose}\ [\textit{OpenTag-}x\ \sigma']] \quad\Rightarrow\quad \sigma\,[\textit{EmptyTag-}x\ \sigma'] \tag{58}$$

Such a system is sufficient for the simple program above; more involved systems could include rules to ensure that parameters are well-formed or even allow progressive construction of the tag's name (although the value of such a feature is questionable).

6 STAXML for PHP

As a proof of concept, we have developed a precompiler for the PHP4 scripting language with full support for the STAXML$_S$ type system and all the mechanisms necessary to support all of the STAXML variants discussed in this paper. In spite of being a highly dynamic language, a meaningful and very useful subset of PHP4 is amiable to static analysis.

This section outlines some of the principles and details of our implementation, “STAXML for PHP”. A web interface to the current version of our implementation is available, and we expect to release the complete source code (including both the web and command-line versions) in the near future.[2]

PHP is not a well formalized language, in the sense that there does not exist a public document which formally defines its syntax, grammar, and semantics; these are described informally by the ubiquitously available PHP Manual [11], but the formal specification is to be had only by understanding the lex, yacc, and C code of its implementation. Our implementation's lexer and parser were derived directly from the php-4.3.0 source; we used the remainder of the Zend engine code to guide our interpretation of the syntax and grammar and inform a few of the basics of our type system.

Our inference algorithm annotates all expressions with a root type: UNSET, NULL, FLOAT, INTEGER, BOOL, ARRAY, OBJECT, UNKNOWN, or STAXML (all “string” values are assigned STAXML types). Each root type has its own set of supplementary data to inform the type; for example, an OBJECT might include an indication of the class from which the object is derived. The STAXML type is supplemented by the actual inferred STAXML type of the expression. The STAXML types of static strings are determined using a simple XML lexer built into our compiler. In both systems, the $\mathcal{E}$ (type error) case is represented by replacing the type stack with a single stack token, *generalstring*, and emitting a warnings; the *generalstring* type is also used when an expression may be any of a set of types which cannot be reconciled with each other.

There is a simple subtype/supertype hierarchy among the root types as well as among STAXML type stacks. The numeric root types have the expected relationship:

$$\text{BOOL} <: \text{INTEGER} <: \text{FLOAT}\ .$$

Naturally, all types are subtypes of UNKNOWN.

Within STAXML types, the particular subtype relationships depend upon which system is being used. For STAXML$_S$, all types are subtypes of *generalstring*; for STAXML, the more involved system presented in Figure 13 is used. More generally, our implementation uses a runtime-configured type resolution and normalization engine; this allows the user to provide a configuration file which specifies the subtype relations and normalizations to apply while checking a program. This makes experimenting with and debugging new STAXML variants much easier than it would be were each system to be hard-coded into the compiler.

6.1 Type Inference for PHP

Type inference is performed by structural inference upon an abstract syntax tree (AST) constructed from PHP program code. An inference algorithm is defined for every kind of abstract syntax node; that algorithm takes as input the node itself and a type environment (a map from variable names to the types associated with those variable names before the execution of this element of syntax), and returns a new type environment (the one in effect when this element of syntax has completed execution) and a type for the value of the block itself (*e.g.*, when inferring the type of expressions). The inference algorithms for each kind of AST node also determine the type of the string (if any) that block would place into the output buffer via `echo`s and appends it to a running accumulator of the output buffer's STAXML type. (The output buffer can be treated like any other variable, having its type stored in the type environment.) The algorithms for several important pieces of PHP4 syntax are presented below; these and others are presented with pseudo-code in greater depth in [3].

At a high level, this inference algorithm is run on the node representing the outermost execution block of a PHP script (code not enclosed within a function or class declaration). When the inference algorithm has completed for the top-level program node, its "echo type" represents the total type of the output stream of the program; it is trivial to check if this type is acceptable (*e.g.*, any form of [*document* []] in STAXML, [] in STAXML$_S$).

For example, the ECHO abstract syntax node represents a command to print some value to the output stream. The inference algorithm for ECHO looks like this (presented in pseudo-code with a C-like syntax):

```
ast_echo_inference(type_env Gamma, ast_echo echo) {
    {type_env Gamma2, staxml_type t} =
            ast_inference(echo->value);
    Gamma2.echo_type =
            type_concat(Gamma2.echo_type, t);
    return {Gamma2, type_unset()};
}
```

Similarly, the SEQ abstract syntax node represents a sequence of two commands or expressions; as in many imperative languages, such sequences can (under some conditions) themselves be treated as values, with the latter expression determining the value and its type. Thus, type inference for SEQ looks like this:

```
ast_seq_inference(type_env Gamma, ast_seq seq) {
    {type_env Gamma2, staxml_type t} =
            ast_inference(Gamma, seq->cmd1);
    return ast_inference(Gamma2, seq->cmd2);
}
```

6.2 Conditionals

Control forks must also be dealt with. Consider the IFTHENELSE abstract syntax node, which has three children: a predicate expression, an "if true" expression/command (possibly empty), and an "if false" expression/command (possibly empty). The implementation of type inference for IFTHENELSE then has the form:

```
ast_ite_inference(type_env Gamma, ast_ite ite) {
    {type_env Gamma2, staxml_type t2} =
            ast_inference(Gamma, ite->predicate);
    {type_env Gamma3, staxml_type t3} =
            ast_inference(Gamma2, ite->iftrue);
    {type_env Gamma4, staxml_type t4} =
            ast_inference(Gamma2, ite->iffalse);
    type_env Gamma5 = reconcile(Gamma3, Gamma4);
    staxml_type t5 = type_lcs(t3, t4);
}
```

Two helper functions do most of the interesting work:

`type_lcs` The LCS of some set of types is their least common supertype. Pairwise least common supertypes are determined using the algorithm discussed above. Naturally, if the only reconciliation between disparate stacks is to use *generalstring*, the appropriate warning/error should be emitted by the compiler.

`reconcile` This operation applies the LCS operation across a pair of type environments. It essentially iterates over the two environments, taking the LCS of their corresponding types for each named variable. If either environment lacks a type for a given variable, the reconciled environment uses the LCS of its type in the other environment and an appropriately-chosen empty-string type. In STAXML$_S$, this is []; in STAXML, we choose [*S* []] to allow it to be more easily reconciled with other non-empty strings.

This handles simple either-or control forks; constructs like SWITCH blocks demand significantly more bookkeeping (particularly when dealing with BREAKs), but in principle work on much the same premise: the ending type environment must represent the reconciliation of the type environment at all possible exit points after all possible execution paths which lead to them.

6.3 Loops

Loops can be challenging for type inference algorithms, particularly when their control flows can be further modified by exceptions like BREAK and CONTINUE commands. Ignoring such exceptions for the moment, consider the WHILE abstract syntax node; it contains two pieces of abstract syntax, the predicate and the loop body. The predicate is executed once, after which there may be zero or more iterations of executing the body followed by the predicate. Our inference algorithm should be able to provide us with a type environment in which every symbol's type is the "tightest" (least) supertype of all types that variable could take on whenever the while loop might exit. Conceptually, we want to find:

```
reconcile(ast_inference(predicate),
          ast_inference(SEQ(predicate,
                            body,predicate)),
          ast_inference(SEQ(predicate,
                            body,predicate,
                            body,predicate)),
          ast_inference(SEQ(predicate,
                            body,predicate,
                            body,predicate,
                            body, predicate)),
          ... );
```

and so on, *ad infinitum*; clearly, however, we cannot explicitly infer and reconcile an infinite set of environments in finite time (let alone tractable time). Instead, it can be demonstrated that iterating the inference algorithm over a finite number of repetitions of the (body, predicate) sequence is always sufficient to infer the type environment after an unbounded number of iterations, where the number of iterations scales linearly by the total execution length of the (body, predicate) sequence.

More precisely, we perform reconciliations between iterations of the loop until the result of the reconciliation operator has reached a *fixed point*, the point at which the type environment has become as general as it needs to be to account for any number of executions of the loop. Ignoring BREAK and CONTINUE, our implementation of inference for WHILE looks like this:

```
ast_while_simple_inference(type_env Gamma,
                           ast_while w) {
    {Gamma1, t1} =
        ast_inference(Gamma, w->predicate);
    do {
        GammaLast = Gamma1;
        {Gamma1,t1} = ast_inference(GammaLast,
                           SEQ(w->body,w->predicate));
        Gamma1 = reconcile(GammaLast, Gamma1);
    } while (Gamma1 != GammaLast)
    return {Gamma1, type_unset()};
}
```

Theorem 6.1. *The fixed-point type inference algorithm for* STAXML *types terminates.*

Proof. A sketch of the complete proof, presented in [3], is as follows:

There are finitely many symbols in any finite block of syntax.

For any given symbol, its type after iteration i of the algorithm must be a supertype of its type after iteration $i-1$ of the algorithm.

The sub/supertype tree has finite depth.

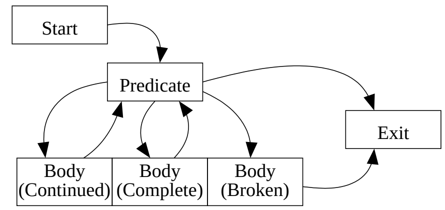

Figure 16: Program flow of the WHILE control construct

Therefore, the type environment can be changed at most nd times, where n is the number of symbols and d is the depth of the type tree. □

Support for Break and Continue The break and continue constructs are dealt with using a more involved inference algorithm and a pair of *environment accumulators* for break and continue, respectively. In principle, the *break environment accumulator* contain the reconciliation of the environments at the points of all BREAKs within a block of code, with the *continue environment accumulator* behaving similarly for CONTINUEs. These values are stored in global variables by the inference algorithm; control structures which rely upon them examine and modify them to capture the desired behavior. Thus, BREAK's inference algorithm is essentially:

```
ast_break_inference(type_env Gamma, ast_break b) {
    atbreak_gamma_accumulator =
            reconcile(atbreak_gamma_accumulator,
                      Gamma);
    return {Gamma, type_unset()};
}
```

and similarly for CONTINUE, where if either argument to `reconcile` is NULL, it simply returns the other argument.

Our full inference algorithm for the WHILE loop is based upon the set of control paths that may be taken through it, illustrated in Figure 16. In essence, we are simply maintaining a local accumulator of every type environment which could exist at a point in execution corresponding to an edge to the "Exit" node.

A full algorithm for the WHILE loop is given in Figure 17. The actual algorithm implemented in our compiler is slightly more complicated in that it also accounts for multi-level BREAKs and CONTINUEs (a single BREAK can be used to exit multiple layers of nested WHILEs, for example). This extension is handled using stacks of `atcontinue_gamma_accumulator`s and `atbreak_gamma_accumulator`s in place of the `preserve_cga` and `preserve_bga` variables, and by adding an appropriate number of iterations up the accumulator stacks to the BREAK and CONTINUE inference algorithms.

```
ast_while_inference(type_env Gamma, ast_while w) {
    {while_gamma_accumulator,t1} =
        ast_inference(Gamma, w->predicate);
    preserve_cga = atcontinue_gamma_accumulator;
    preserve_bga = atbreak_gamma_accumulator;
    atcontinue_gamma_accumulator = NULL;
    atbreak_gamma_accumulator = NULL;
    do {
        hold_wga = while_gamma_accumulator;
        hold_bga = atbreak_gamma_accumulator;
        {Gamma2, t2} = ast_inference(
                reconcile(hold_wga,
                    atcontinue_gamma_accumulator),
                w->body);
        {Gamma3, t3} = ast_inference(Gamma2,
                                    w->predicate);
        while_gamma_accumulator =
                reconcile(hold_wga, Gamma2,
                        atbreak_gamma_accumulator);
    } while ((while_gamma_accumulator == hold_wga) &&
            (atbreak_gamma_accumulator == hold_bga));
    atcontinue_gamma_accumulator = preserve_cga;
    atbreak_gamma_accumulator = preserve_bga;
    return {reconcile(while_gamma_accumulator,
                    atbreak_gamma_accumulator),
            type_unset()};
}
```

Figure 17: STAXML for PHP inference algorithm for WHILE loops

The algorithm for DO is similar; the principal differences are that DO executes the body before the predicate is evaluated and that CONTINUE causes the body of the DO block to be re-started without the predicate being re-evaluated.

Because FOR and FOREACH loops (the two remaining loop constructs in PHP) are transformed into WHILE loops in the abstract syntax, they are handled by the above algorithm.

6.4 Functions

PHP functions are by their nature polymorphic even in the native PHP type system; their declarations include no type annotations, so they can be called using any type of arguments (and can consequently return any type of value and have any type of side-effect upon the output stream's type and the global type environment via access to global variables). What's more, function arguments can be passed either by value or by reference, and the by-reference behavior can be activated by either or both of the function's declaration and the syntax of a function call itself (although the forthcoming PHP5 release appears to be omitting support for function calls requesting the call-by-reference semantic).

In the common case in which there is no recursion and no global variables are used, the algorithm is straightforward. For each declared function, we construct on demand a map from argument types to a tuple of return type, echo type, and types of arguments at return points (in case any are called by reference); each time a function call is encountered, it will either draw its type information from the map (if that function with those particular argument types has been encountered before) or it will compute it using syntax-directed inference upon the function body. Handling reads from and writes to global variables would requires that we also index results based upon the types of those global variables at call-time and track their types at return time; this extension is not yet included in our implementation, although some of necessary code is already present in the implementation.

The returned types and type environments are determined by identifying all possible exit points from the body of the function and taking the reconciliation of the types and type environments at each of those points, *i.e.*, at every RETURN and at the end of the function body. Only the RETURN statement can give the function a return value, so if a function contains any RETURNs of values and can also reach the end of its body without RETURNing, a warning is thrown (as the function might return a typed value but also might not return anything at all).

Given the inherent difficulty of type inference for recursive polymorphic functions [7], we replace inference in cases of recursion with validation of a typing assumption: if ever a function is found to be able to recursively call itself (whether directly or indirectly), that function must be "pure" (it must contain no accesses to global or static variables and must have no by-reference arguments) and its echo and return types must both correspond with some pre-defined STAXML type which varies with the particular system; for STAXML$_S$, the defaults for both are []; in STAXML, the defaults for both are [*content* []]. If a type inference pass of the function (relying upon this assumption) produces a result which disagrees with the assumption (*i.e.*, if our assumption is disproven), a type error is thrown and we type the function as returning and echoing *generalstring*.

6.5 Limitations

Our implementation has minimal or no support for several PHP4 constructs:

Arrays and Records The "array" mechanism in PHP is used to implement both arrays (where all members are of a single type) and records (where each keyed member may have a distinct type and that key consistently corresponds with that type). Since the language includes no type annotations to indicate the programmer's intent, our inference algorithm assumes that the intended semantic is that of the array, *i.e.*, the type of the array itself is "ARRAY OF x" where x is the LCS of all values assigned as members to that array.

Classes and Objects The implementation currently includes no support for classes, *i.e.*, it does not infer types for object properties or methods, and simply gives up when it encounters them. Support for classes and object is largely a matter of additional software engineering to perform the requisite bookkeeping.

Variable Variables While PHP does not include an explicit notion of pointers, it does support another kind of dereferencing with "variable variables", by which a variable's value itself can be used as a variable name. (For example, the program `$x= ''y''; $y=1; echo $$x;` will cause "1" to be printed to the program's output stream.) This is a particularly thorny form of the aliasing problem, since aliases are derived not through a flat address space but through the lexical space of variable names as strings. While our pre-compiler does attempt to aggressively propagate static values for inference purposes, when it is not possible to find a static value used in a variable-variable we treat the result as having type UNKNOWN (which, when concatenated with a string, becomes [*generalstring* []]).

The forthcoming PHP5 release includes features which will both simplify and complicate the task of static inference for PHP scripts. While removal of on-demand by-reference calling semantics and the introduction of type hints (type declarations for function parameters) simplify function call type inference, the expansion of the object model to include member protection, abstract classes, and interfaces promises to complicate inference support when objects are used; the introduction of exceptions to the programming model will also demand support for inter-procedural inference steps (currently not supported).

6.6 Runtime Library Functions

Naturally, establishing appropriate types for the runtime library is an important part of making the STAXML type systems usable for real programs. For much of the PHP runtime environment, this is not terribly hard; operations which do not return strings are already easily typed, and most operations which return strings return them with no guarantees upon what kinds of characters or substrings may appear within them, and thus must be typed as [*generalstring* []].

There are a few exceptions. The `htmlentities` and `htmlspecialchars` functions always return strings which do not contain XML/HTML control characters (like angle brackets), and can thus be treated as plain text.

PHP's approach to initializing arrays is also worth special attention; the syntax used to create an array looks very much like a function call, but is handled internally by the interpreter as a distinct mechanism. For example, the expression `Array("<doc>This", "<doc>That", "<doc>The Other Thing");` returns an array with three elements corresponding with what are (syntactically) the three arguments to the Array "function". This is easy enough to handle as a special case of type inference when the function name is "Array"; in this case, STAXML$_S$ or STAXML would both infer this expression to have the type "ARRAY of [*STag-doc* []]", which is the common supertype (and in this case, the precise type) of all three arguments.

The mechanism by which our implementation stores types for runtime library functions is the same mechanism used to cache inference results for program functions. When the compiler starts up, an initialization function populates the cache with records for each standard library function. By augmenting the cache with the ability to use *wildcard* types in the argument-type index, we are able to include in the cache particular "tight" return/echo types for functions when called with particular argument types for which their behavior is more restrictive and failsafe general types to catch all other cases.

6.7 Experience

We have tested STAXML for PHP on a number of home-brewed PHP scripts of up to three thousand lines in length. Our ability to test third-party code has been restricted by our implementation's thusfar limited support for compound data types (arrays and classes/objects) and their popularity with PHP module developers; we are currently working to remedy this, and expect to have much broader results to report on in the near future.

7 Conclusions and Future Work

STAXML is a simple and computationally tractable type system which enforces XML-like well-formedness upon programs which construct XML documents piecemeal from individually unbalanced fragments.

The STAXML for PHP pre-compiler is currently being prepared for public release under the GNU GPL. We are currently completing support for type inference upon compound data structures (arrays, record, and class/objects) and tweaking support for the STAXML$_A$ and STAXHTML type systems. A web gateway to the current version of the precompiler is already available.

We are also exploring the applicability of similar techniques to validating imperative construction of other kinds of structured strings commonly used in Internet-enabled applications (*e.g.*, SQL queries) and at the possibility of inferring interesting implicit properties of such strings (*e.g.*, the safety or idempotence of such a query).

Acknowledgements

SDG.

Notes

1. This enhancement can be equivalently implemented by converting every stack element type into 32 types corresponding with all possible values of the 5 flags and correspondingly multiplying the number of normalization rules, modifying them to throw type errors in the appropriate cases.

2. All STAXML resources can be found on the web at `http://cs-people.bu.edu/artdodge/sc.php/staxml`

References

[1] Franz Baader and Tobias Nipkow. *Term* Re*writing and* All That. Cambridge University Press, 1998.

[2] Claus Brabrand, Anders Moller, and Michael I. Schwartzbach. Static validation of dynamically generated HTML. In *PASTE 2001*, Snowbird, UT, 2001. ACM.

[3] Adam D. Bradley. *A Type-Disciplined Approach to Developing Resources and Applications for the World-Wide Web*. PhD thesis, Boston University, 2004.

[4] Tim Bray, Jean Paoli, and C. M. Sperberg-McQueen. Extensible markup language (XML) 1.0, 2000. http://www.w3.org/TR/REC-xml.

[5] H. Hosoya and B. C. Pierce. Xduce: A typed XML processing language. In *Int'l Workshop on the Web and Databases (WebDB)*, Dallas, TX, 2000.

[6] Martin Kempa and Volker Linnemann. Towards valid XML applications. In *International Conference on Advances in Infrastructure for e-Business, e-Education, e-Science, and e-Medicine on the Internet (SSGRR)*, L'Aquila, Italy, 2002.

[7] Assaf J. Kfoury and Santiago M. Pericas-Geertsen. Type inference for recursive definitions. In *Proc. 14th Ann. IEEE Symp. Logic in Comput. Sci.*, pages 119–128, July 1999.

[8] David A. Ladd and Christopher Ramming. Programming the web: An application-oriented language for hypermedia service programming. In *Fourth International World Wide Web Conference (WWW4)*, Boston, MA, December 1995.

[9] Santiago M. Pericas-Geertsen. *XML-Fluent Mobile Agents*. PhD thesis, Boston University, 2001.

[10] Anders Sandholm and Michael I. Schwartzbach. A type system for dynamic web documents. In *POPL 2000*, Boston, MA, 2000.

[11] Stig Sæther Bakken, Alexander Aulbach, Egon Schmid, Jim Winstead, Lars Torben Wilson, Rasmus Lerdorf, Zeev Suraski, and Andrei Zmievski. PHP documentation, 2003. http://www.php.net/manual/.

[12] P. Thiemann. Modeling HTML in Haskell. In *Practical Applications of Declarative Programming (PADL '00)*, Boston, MA, January 2000.

[13] P. Thiemann. A typed representation for HTML and XML documents in Haskell. Technical report, Universität Freiburg, February 2001. Revised version to appear in Journal of Functional Programming.

[14] W3C. XHTML™ 1.0 the extensible hypertext markup language (second edition), 2002. http://www.w3.org/TR/xhtml1/.

[15] W3C. Extensible markup language (XML), 2003. http://www.w3.org/XML/.

[16] Malcolm Wallace and Colin Runciman. Haskell and XML: Generic combinators or type-based translation? In *Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming (ICFP'99)*, volume 34–9, pages 148–159, N.Y., 27–29 1999. ACM Press.