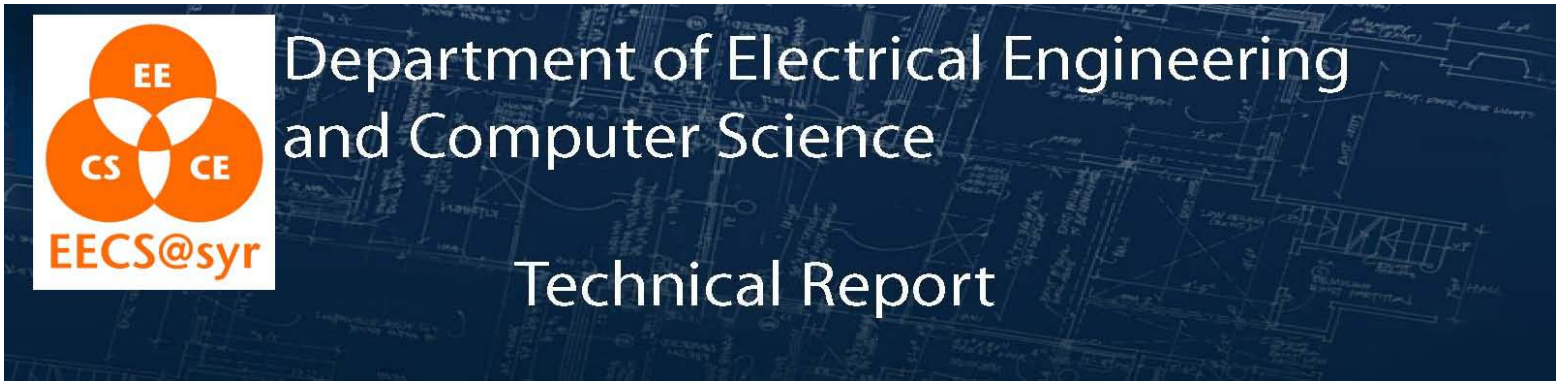
10-22-2010

Strong (X)HTML Compliance with Haskell's Flexible Type System

Paul G. Talaga
Syracuse University

Steve J. Chapin
Syracuse University



SYR-EECS-2010-04 Oct. 22, 2010

Strong (X)HTML Compliance with Haskell's Flexible Type System

Paul G. Talaga pgtalaga@syr.edu
Steve J. Chapin chapin@syr.edu

ABSTRACT: We report on the embedding of a domain specific language, (X)HTML, into Haskell and demonstrate how this superficial non-context-free language can be represented and rendered to guarantee World Wide Web Consortium (W3C) compliance. Compliance of web content is important for the health of the Internet, accessibility, visibility, and reliable search. While tools exist to verify web content is compliant according to the W3C, few systems guarantee that all produced content is compliant. We present *CH-(X)HTML*, a library for generating compliant (X)HTML content by using Haskell to encode the nontrivial syntax of (X)HTML set forth by the W3C. Any compliant document can be represented with this library, while a compilation error will occur if non-compliant markup is attempted. To demonstrate our library we present examples and performance measurements.

KEYWORDS: W3C Compliance, Haskell, Web Development

Syracuse University - Department of EECS,
4-206 CST, Syracuse, NY 13244
(P) 315.443.2652 (F) 315.443.2583
http://eecs.syr.edu

Strong (X)HTML Compliance with Haskell's Flexible Type System

Paul G. Talaga
Syracuse University
pgtalaga@syr.edu

Steve J. Chapin
Syracuse University
chapin@syr.edu

Abstract

We report on the embedding of a domain specific language, (X)HTML, into Haskell and demonstrate how this superficial non-context-free language can be represented and rendered to guarantee World Wide Web Consortium (W3C) compliance. Compliance of web content is important for the health of the Internet, accessibility, visibility, and reliable search. While tools exist to verify web content is compliant according to the W3C, few systems guarantee that all produced content is compliant. We present *CH-(X)HTML*, a library for generating compliant (X)HTML content by using Haskell to encode the non-trivial syntax of (X)HTML set forth by the W3C. Any compliant document can be represented with this library, while a compilation error will occur if non-compliant markup is attempted. To demonstrate our library we present examples and performance measurements.

1 Introduction

Conformity of web content to the World Wide Web Consortium's (W3C) standards is a goal every web developer should aspire to meet. Conformity leads to *increased visibility* as more browsers can render the markup consistently, *increased accessibility* for disabled users using non-typical browsing styles [8], *more reliable Internet search* by presenting search engines with consistent page structures [14], and in some cases *compliance with legal requirements* [3, 6, 20, 29].

Unfortunately the majority of web content is non-compliant, with one study finding 95% of pages online are not valid [12]. Not surprisingly, the majority of web frameworks do not guarantee generated content is compliant. Popular internet browsers perpetuate the problem by creatively parsing and rendering invalid code in an attempt to retain users.

While tools exist to check validity of static content, few systems exist that claim strong validity of *all* produced content. With dynamic web applications, it is harder to guarantee validity due to the dynamic nature of their inputs. Assuring compliance for specific inputs is possible, but proving compliance for all inputs is analogous to proof by example. Web frameworks using Model-View-Controller design practices provide some assurances based on compliant templates, but it remains easy for an unknowing developer or specific user input to break this compliance. Such deficiencies in frameworks can have security consequences as well [23]. Rather than make it easy for developers to produce invalid content, frameworks should make it impossible to be non-compliant.

We view an (X)HTML page as a tree structure with HTML tags representing nodes. Tag attributes are properties of a node. Inner tags are children of parent nodes and each node can contain any number of children, when allowed. Any language could represent this structure, but using Haskell's multiple parameter and functional dependencies of type classes allows simpler syntax for the developer while a more complex dependency scheme exists beneath.

1.1 Contributions

We present *CH-(X)HTML*, a Haskell library for building (X)HTML content with strong W3C compliance. By using Haskell's recursive types, multiple parameter and functional dependency of type classes, web content is built by separating structure from content in a typed tree data structure way, much like the underlying (X)HTML. The resulting structure can be stored, manipulated, or serialized to a standard W3C compliant textual representation.

The remainder of the paper is structured as follows. We analyze and categorize commonalities between different W3C (X)HTML specifications in Section 2, identifying requirements a W3C compliant producing system must possess. Section 3 provides an overview of *CH-(X)HTML* and discusses how it is able to enforce

the W3C specifications while being easy to use. Sample code is provided showing the use of the library, followed by a performance evaluation in Section 3.3 Related work and our conclusion are in Sections 4 and 5 respectively.

2 W3C Compliance

The W3C has set forth numerous variants of specifications of HTML and XHTML, with more on the way in the form of HTML5. Examples include HTML 3.2, HTML 4.01 Strict, and XHTML 1.00 Transitional. While conformance to a specific document type definition (DTD) is our goal, identifying commonalities will assure easy conversion to any HTML DTD. For example, the difference between HTML 4.01 Strict and HTML 4.01 Transitional is merely the allowance of certain tags. Likewise, HTML 4.01 Frameset and XHTML 1.00 differ in their document type: SGML and XML respectively [16].

We have identified five classes of common requirements between the different (X)HTML DTDs based on Thiemann's work [25]. A system capable of supporting all requirement classes should be able to include support for all requirements in any of the W3C specifications if fully implemented. These classes include the following:

1. Well-formed
2. Tag-conforming
3. Attribute-conforming
4. Inclusion & Exclusion conforming
5. Tag ordering

Well-Formed: An (X)HTML document is well-formed if all tags have an appropriate ending tag when needed. All attributes have the form `attribute="value"` inside a tag. All characters should be in the correct context. For example, all markup characters should only be used for markup including `<`,`>`,`&`,`"`.

Tag-conforming: An (X)HTML document is tag-conforming if all tags are defined and valid within that DTD. No browser specific tags should be used.

Attribute-conforming: An (X)HTML document is attribute-conforming if all attributes names are allowed for that specific tag. For example, the `p` tag can not contain an `href` attribute.Similarly, the value type of every attribute matches its DTD description.

Inclusion & Exclusion: An (X)HTML document obeys inclusion & exclusion if the nesting of all tags follow the specific DTD. For example, in HTML 4.01 no `a` tag can be a descendant of another `a` tag. Similarly, the `tr` tag requires a `td` tag to be its child. While SGML, of which HTML is a member, allows deep nesting rules, XML does not [16]. XML can specify what children are allowed, but not grandchildren or beyond. Thus, the XHTML 1.0 specification recommends the inclusion & exclusion of tags, but can not require it. We feel that since XHTML is fully based on HTML this requirement is important and should be enforced. In support, the W3C online validator marks inclusion & exclusion problems in XHTML as errors. The draft HTML5 specification broadens nesting rules by restricting *groups* of tags to be children [5]. For example, an `a` tag in HTML5 must not contain any *interactive content*, of which 15 tags are members.

Tag ordering: An (X)HTML document obeys tag ordering if sibling tags are ordered as described in their DTD. As an example, the `head` tag must precede the `body` tag as children of the `html` tag.

3 CH-(X)HTML

Our system is built as an embedded domain-specific language, implemented in Haskell, capable of embodying many requirements set forth by the W3C. The use of a strongly typed language guarantees compliance of the application at *compile* time, while allowing easy representation of the embedded language. Any strongly typed language could be used for such a system, but Haskell's multiple parameter and functional dependency type classes cleans up the syntax for the developer.

CH-(X)HTML is available for download or review at `http://fuzzpault.com/chxhtml` Only Xhtml 1.0 Strict [16] is currently supported at this time.

3.1 Implementation Overview & Example

CH-(X)HTML's design is outlined through a series of refinements presented below. Code examples are meant to convey design methods, not produce fully correct HTML.

At its core, *CH-(X)HTML* uses ordinary Haskell types to implement a recursively defined tree data structure representing the (X)HTML document. Each node in the tree represents a tag, with inner tags stored as children of the parent. Depending on the tag, the node may have none, or a variable number of children. Tag attributes are stored with each node. Character data is inserted using an extra constructor. An example of this scheme is given:

```
data Ent = Html Attributes [Ent] |
           Body Attributes [Ent] |
           P Attributes [Ent] |
           Br Attributes |
           Cdata String | ...
Attributes = [String]
```

When the data structure has been constructed and is ready to be serialized, a recursive function `render` traverses the structure, returning a string containing tags and properly formatted attributes and values. All character data (CDATA) is HTML escaped before rendering preventing embedding of HTML markup. This approach, thus far, guarantees well-formed and tag-conforming documents when rendered.

We add tag-specific attribute types to enforce attribute conformance.

```
data Ent = Html [Att_html] [Ent] |
           Body [Att_body] [Ent] |
           P [Att_p] [Ent] |
           Br [Att_b] |
           Cdata String | ...
--
data Att_html = Lang_html String |
                Dir_html String  | ...
data Att_body = Lang_body String |
                Dir_body String  |
                Onload_body String | ...
...
```

Thus far any tag can be a child of any other. For inclusion & exclusion conformance we use new data types representing the context of those tags. The same serialized tag can now be inserted by any number of constructors depending on context. For example the following code correctly prohibits nesting of the `a` tag.

```
data Ent = Html Att_html [Ent2]
data Ent2 = Body Att_body [Ent3]
data Ent3 = A3 Att_p [Ent_no_a] |
            P3 Att_p [Ent3] |
            Br3 Att_b |
            Cdata3 String | ...
data Ent_no_a = P_no_a Att_p [Ent_no_a] |
            Br_no_a Att_b |
            Cdata_no_a String | ...
-- Attributes same as above
```

By explicitly describing what type can be a child of an `a` tag we prevent any nesting issues no matter what the depth. A compile-time error will be thrown if an invalid nesting situation is attempted.

Writing (X)HTML content using these complex constructors becomes unwieldy quite quickly. By using multi-parameter type classes and functional dependencies we can hide this complexity while still retaining the compile-time guarantees. We construct a type class per tag such that a function correctly returns a constructor of the correct type based on context. The following example shows the type class for specifying the `p` tag.

```
class C_P a b | a -> b where
    p :: [Att_p] -> [b] -> a
instance C_P Ent3 Ent3 where
    p at r = P_1 at r
instance C_P Ent_no_a Ent_no_a where
    p at r = P_2 at r
```

The class instance used is determined by the context the function is called in, which determines what type children it may have provided by the functional dependency of classes. Thus, as long as the root of the recursive structure has a concrete type all children will be uniquely defined. Nesting errors manifest themselves in compile-time class instance type errors.

A similar type class system is used to simplify attribute specification.

The previous design methods produce a library whose output is well-formed, tag-conforming, attribute-conforming, and inclusion/exclusion conforming. Tag ordering is not enforced due to a list representation of children nodes. While ordering could be enforced using some other means, in practice tag ordering is not easily violated, while the list of children nodes allows existing Haskell functions to be used for easy manipulation.

3.2 Library Usage

Building an (X)HTML document is done by constructing the recursive data structure and serializing it using the `render` or `render_bs` functions. Static or dynamic content can be served with any number of Haskell web servers such as HAppS [4], Happstack [15], MoHWS [19], turbinado [17], SNAP [13], or via executable with CGI [2] [10] or FastCGI [24] [11] Haskell bindings. *CH-(X)HTML* can be used anywhere a `String` type containing (X)HTML is needed in Haskell. For speed and efficiency the `render_bs` function returns a lazy ByteString representation suitable for CGI bindings.

All HTML tags are represented in lower case with an underscore _ before or after the tag text. Before assigns no attributes, while after allows a list of attributes. Tags which allow children then take a list of children.

Attributes are represented in lower case as well, but suffixed with `_att`. This assures no namespace conflicts. Assigning an attribute which does not belong results in a compile-time class instance error.

Figure 1 exhibits the obligatory Hello World page where `result` holds the resulting serialized HTML as a string.

For a more through description of the *CH-(X)HTML*'s usage see the demo.hs included with the library source.

3.3 Library Performance

To gauge our library's performance against similar dynamic HTML creation systems, we implemented a dynamic page in each of four systems: Text.Html, Text.XHtml, PHP, and *CH-(X)HTML*. Text.Html and

```
page name = _html [_head [_title [pcdata "Hello " ++ name]],
                _body [_h1 [pcdata "Hello " ++ name ++ "!"],
                                          _hr,
                       _p [pcdata "Hello " ++ name ++ "!"],
                      ]
                  ]
result :: String
result = render (page "World")
```

Figure 1: Hello World implementation in *CH-(X)HTML*

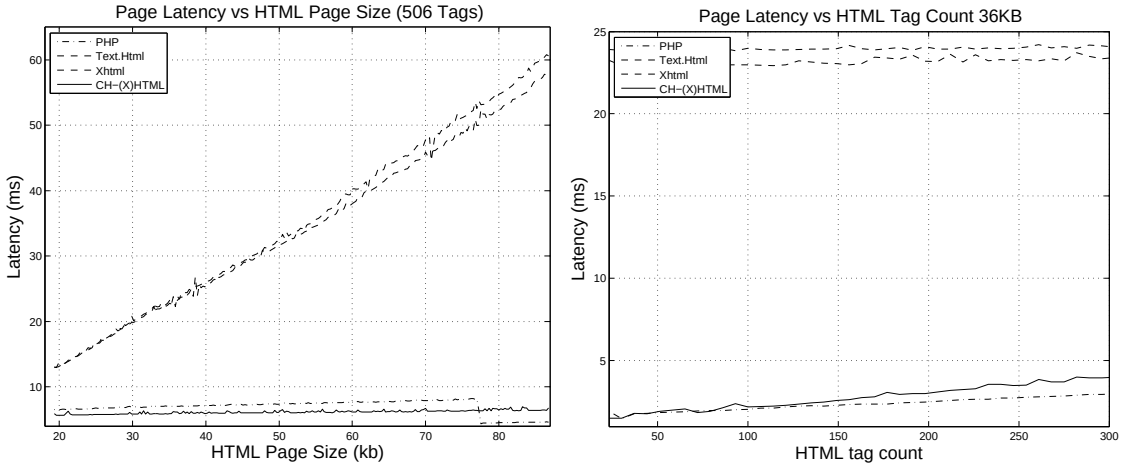


Figure 2: Latency comparison between *CH-(X)HTML* , Text.Html, Text.XHtml, and PHP

Text.XHtml are combinator libraries in Haskell used for building HTML content while PHP is a popular web scripting language.

The page consisted of a variable number of nested `div` segments containing text and a variable size multiplication table limited to 10 columns. Both div text length and table size are controllable via GET parameters. This allows independent control of total page length and the number of tags used. In addition to general speed, we were interested in how the underlying tree structure impacted performance. By finding a relationship between the two parameters it was possible to produce identical length pages while varying the number of tags, thereby measuring structure and logic latency while ignoring any effect page length may have. Conversely the number of tags could be held constant while the page size was changed.

Each dynamic page was served from a Fedora Core 11 server with an AMD Athlon 64 X2 processor, 2GB of RAM, and 1000BaseT Ethernet connection. An Apache (httpd 2.2) web daemon served all content using PHP 5.2.11 or a FastCGI 2.4.6 interface for Haskell. GHC 6.10.3 was used to compile the Haskell code to executables used by FastCGI. ApacheBench (ab 2.3) [1] was used to measure the latency of the dynamic applications on the same machine.

For a representative sample of real-world performance, a collection of random web pages were sampled from Google searches. 30,000 (X)HTML documents were obtained with an average size of 36KB and an average tag count of 506. This drove our testing by varying page size and tag counts around these values.

Figure 2 presents two latency plots showing the relation between latency and page size, as well as latency and tag count. For each set of parameters, 500 requests were made and their latencies averaged. As can be seen *CH-(X)HTML* performs on par with PHP, significantly faster than the other Haskell libraries.

Text.Html and Text.Xhtml both exhibit similar behavior as page size is increased, most likely due to their use of the `String` datatype implemented as a linked list of characters [21]. *CH-(X)HTML* uses `ByteString` type internally to manipulate text and return the result, in this case, to FastCGI.

As tag count increases we see a slight slowdown in both PHP and *CH-(X)HTML*, unlike Text.Html and Text.Xhtml which stay consistently slow.

CH-(X)HTML 's use of `ByteString` clearly puts it in a performance category comparable to production dynamic web content generators such as PHP, while the structure representation does not effect latency significantly. Existing `String` based Haskell libraries are at a clear disadvantage when performance is considered.

4 Related Work

Constructing web content by means of a DOM-like data structure isn't new, but libraries guaranteeing near or full HTML validity are scarce. Many HTML libraries use HTML like syntax, allowing easy construction of pages for the developer, with little guarantees to the validity of the output. Peter Thiemann's work on W3C compliance is the closest in the Haskell WASH/CGI suite [27] [28] [25], which includes a HTML & XML content production system using types to enforce some validity. The use of element-transforming style in the library allows Haskell code to look similar to HTML while still being valid Haskell source. The author documents different classifications of validity, which our analysis in Section 2 is based on, followed by a discussion of enforcement of those classifications in his system. The Inclusion & Exclusion issue is raised and discussed briefly in his 2002 work, concluding the type class system is unable to handle inclusion & exclusion in their implementation due to the inability to handle disjunctions of types. As a result, their library does not support inclusion or exclusion with the excuse of extreme code size, difficulty in usability, and a lack of strict guidelines for inclusion & exclusion in the XHTML specification. They further mention compilation errors are difficult to understand due to the complex implementation of element-transforming style and multi-parameter types. By not using element-transforming style, *CH-(X)HTML* is simpler to use, presents simpler compilation error messages, and can enforce inclusion & exclusion.

Further work explores an alternate way of dealing with the inclusion & exclusion issue in Haskell [26]by way of proposed extensions providing functional logic overloading, anonymous type functions, and rank-2 polymorphism. With these they are able to accurately encode and enforce the inclusion & exclusion properties specified in the DTD. A strong symmetry exists between our work and the suggested extensions. The ability to embed regular expressions on types is analogous to our generous use of recursive types. While extending the type system further may lead to more enhancements, *CH-(X)HTML* can be used currently in GHC without any additional extensions.

HSXML is an XML serialization library for functional languages [18]. It is part of a larger effort to parse XML into S-expressions in functional languages such as Scheme and Haskell, with HSXML preforming the reverse. S-expressions are a natural way of representing nested data with its roots in Lisp, thereby guaranteeing a well-formed and tag-conforming document. The library's current implementation can handle Inline vs Block context restrictions, but no other inclusion/exclusion restrictions are enforced.

A common Haskell HTML library is Text.Html [7] and relative Text.XHtml used above, which uses element-transforming style to build pages. Produced content is well-formed and tag-conforming due to their structured building method and HTML escaping of text content. Any attribute can be added to any tag, thus not being attribute-conforming. All tags are of the same type and can be added in any order leading to tag ordering and inclusion/exclusion violations.

Element-transforming style present in many of the previously mentioned libraries can lead to difficulties when building dynamic pages. Rather than represent children nodes as a Haskell list, they are represented in some other hidden form not easily manipulated with normal Haskell list processing functions. We chose the list type for ease of implementation, at the cost of tag ordering compliance.

Separating structure from content in a web setting is advantageous for security as well. Robertson & Vigna [22] explore using a strongly typed system for HTML generation as well as producing SQL queries in the web application. Their goal is to increase security by preventing injection attacks targeting the ad-hoc mixing of content and structure by representing structure in a typed way and filtering inserted content. Thus, the client or SQL server's parser will not be fooled by the attempted injection attack. Our work similarly mitigates injection attacks but does not address web application vulnerabilities relating to a database.

XMLC for Java allows an application developer to manipulate a DOM structure obtained from parsing a HTML or XML template file [9]. Manipulation of the DOM is therefore similar to DOM manipulations in JavaScript. When all transformations are complete the DOM is serialized and sent to the user. XMLC does not restrict operations which would result in invalid content being sent to the user.

5 Conclusion

We have shown how strong (X)HTML W3C compliance can be achieved by Haskell while performing on par with more mature dynamic (X)HTML production systems. We generalize the W3C (X)HTML specifications into five classes of requirements a web production system must be able to enforce to produce compliant output. The inclusion & exclusion nesting requirement of nearly all (X)HTML DTD's has proven difficult to enforce and thus ignored by web production libraries. Our (X)HTML library, *CH-(X)HTML* , is able to enforce four of the five classes of requirements, including inclusion & exclusion, by using recursive types. Use of the library is straightforward due to multi-parameter type classes and functional dependencies allowing a coding style similar to straight

(X)HTML, while guaranteeing strong compliance for all produced content.

References

[1] Apache http server benchmarking tool, http://httpd.apache.org/docs/2.0/programs/ab.html.

[2] The common gateway interface, http://hoohoo.ncsa.illinois.edu/cgi/.

[3] The disability discrimination act (dda), http://www.direct.gov.uk/en/DisabledPeople/RightsAndObligations/DisabilityRights/DG_4001068.

[4] Happs, http://happs.org/.

[5] Html5, http://dev.w3.org/html5/spec/Overview.html.

[6] Policies relating to web accessibility, http://www.w3.org/WAI/Policy/.

[7] Text.html, http://hackage.haskell.org/package/html.

[8] Web content accessibility guidelines 1.0, http://www.w3.org/TR/WCAG10/.

[9] Xmlc, http://xmlc.enhydra.org.

[10] B. Bringert. cgi: A library for writing cgi programs, http://hackage.haskell.org/package/cgi.

[11] B. Bringert and Lemmih. fastcgi: A haskell library for writing fastcgi programs, http://hackage.haskell.org/package/fastcgi.

[12] S. Chen, D. Hong, and V. Y. Shen. An experimental study on validation problems with existing html webpages. In *International Conference on Internet Computing*, pages 373–379, 2005.

[13] G. Collins, D. Beardsley, S. yu Guo, and J. Sanders. Snap: A haskell web framework, http://snapframework.com/.

[14] D. Davies. W3c compliance and seo, http://www.evolt.org/w3c-compliance-and-seo, oct 2005.

[15] M. Elder and J. Shaw. Happstack, http://happstack.com/index.html.

[16] W. H. W. Group. Xhtml 1.0: The extensible hypertext markup language (second edition). http://www.w3.org/TR/xhtml1/, http://www.w3.org/TR/xhtml1/, aug 2002.

[17] A. Kemp. Turbinado, http://wiki.github.com/alsonkemp/turbinado.

[18] O. Kiselyov. Hsxml: Typed sxml, http://okmij.org/ftp/Scheme/xml.html#typed-SXML.

[19] S. Marlow and B. Bringert. Mohws: Modular haskell web server, http://hackage.haskell.org/cgi-bin/hackage-scripts/package/mohws.

[20] T. Moss. Disability discrimination act (dda) & web accessibility, http://www.webcredible.co.uk/user-friendly-resources/web-accessibility/uk-website-legal-requirements.shtml.

[21] B. O'Sullivan, J. Goerzen, and D. Stewart. *Real World Haskell*. O'Reilly, 1 edition, 2009.

[22] W. Robertson and G. Vigna. Static Enforcement of Web Application Integrity Through Strong Typing. In *Proceedings of the USENIX Security Symposium*, Montreal, Canada, August 2009.

[23] R. H. (RSnake). Xss (cross site scripting) prevention cheat sheet, http://ha.ckers.org/xss.html.

[24] R. Saccoccio et al. Fastcgi, http://www.fastcgi.com/drupal/.

[25] P. Thiemann. A typed representation for html and xml documents in haskell. *Journal of Functional Programming*, 12:2002, 2001.

[26] P. Thiemann. Programmable type systems for domain specific languages, , 2002.

[27] P. Thiemann. Wash/cgi: Server-side web scripting with sessions and typed, compositional forms. In *Practical Aspects of Declarative Languages: 4th International Symposium, PADL 2002, volume 2257 of LNCS*, pages 192–208. Springer-Verlag, 2002.

[28] P. Thiemann. An embedded domain-specific language for type-safe server-side web-scripting. *ACM Transactions on Internet Technology*, 5:1533–5399, 2005.

[29] A. Wittersheim. Why comply? the movement to w3c compliance, http://ezinearticles.com/?Why-Comply?-The-Movement-to-W3C-Compliance&id=162596.