The Parallel Web:
How to write a Web Service
in a Functional Programming Language

Henning Thielemann

2nd June 2007

Abstract

The parallel web is an online translation service which modifies the hyper links of a translated document such that referred documents become translated in turn. This creates the impression that you navigate through a parallel web which has the same structure like the "original" WWW but contains translated text.

This article describes an implementation using the pure functional programming language Haskell. That is the HTML processing and the (toy) translators, as well as the used packages for the web server and the HTML document download are written in Haskell. We show how the components can be separated cleanly and safely due to lazy evaluation and strong static typing. This yields a very flexible and secure, yet efficient program.

1 Introduction

Imperative programming languages like C, its derivatives, and scripting languages, are the usual choice for today's software systems. The currently widespread CPUs conform to the imperative programming paradigm and allow a fast execution of imperative programs. Although web services require a high level of security, the efficiency aspect seems to be strong enough that C was chosen for wide-spread web servers like Apache.

Compared to imperative programming languages, functional languages are usually higher level languages, aiming at higher security and maintainability. The presented web service is implemented in **Haskell** [PJ98, Dau04], a functional language that has become very popular in the recent past. The term *Functional Programming* [Hug89, Hud89] denotes a kind of program flow that is different from the *imperative* one. The program flow is independent from a particular type system and from whether programs can be compiled or not. Indeed **Haskell** is *strongly typed*, including statically checked *polymorphic typing*, and it can be both interpreted [JR04] and compiled [HHPPJ04]. Because there is no state, no mutable variables can be declared. This simplifies working in an interactive interpreter. In view of the variety of machine oriented imperative languages and imperative scripting languages, **Haskell** is one of the rare languages that provides both strong static typing and interactive function execution.

Although not recognized as a language for web applications there are a lot of web related libraries and applications including a web server [Mar00], frameworks for CGI applications [Thi02, Bri07], and libraries for HTML processing, [Wal07, Sch07, Mit07]. For a more complete list, see

`http://www.haskell.org/haskellwiki/Applications_and_libraries/Web_programming.`

2 Functional programming basics

Today functional concepts are integrated into almost every imperative programming language. They allow for structured and safe programming. A function is a part of a program with some declared input and output.

Most imperative languages allow for bypassing the input/output interface by the use of global variables and by manipulation (update) of input objects. In contrast to that, **Haskell** disallows or at least strongly discourages that. This makes things more predictable: If you apply a function to the same argument values it will always result in the same value. This property is called *referential transparency*. It allows for effective testing and even rigorous proofs. If you found (by tests or by proofs) that a function returns the wanted output for a particular input, then the function will reliably return that result for future calls with the same input.

In this framework it seems to be impossible to communicate with the outer world or to work with, say random number generators. The function `getLine`, which returns a line of text entered by the user, can obviously return different values each time called. The same applies to the function `random`, which returns a random object of a particular type for each call. It was a big breakthrough to formalize this kind of functions (and others) by the notion of *monad*s. We will consider this issue in more detail in section 3.3.

The functional approach allows features that can be imagined hardly for imperative programming languages. One such feature is *lazy evaluation*. It means that function arguments and parts of data structures are only computed if they are needed. Once computed, they are usually stored for later access, or flushed by a garbage collector, if they become unaccessible. Because of lazy evaluation a list may formally contain infinitely many elements. This poses no problem since a terminating algorithm will be able to process only finitely many of them. We will represent our data as potentially infinite list of characters, as list of HTML tags, and as tree representing the HTML document structure. Data that is read from a web server could be dynamically generated, and could be indeed infinite. We can cope with this case although we organize our program as a sequence of transformations of entire lists or trees!

Try yourself in an interactive Haskell environment like `hugs` or `ghci` from the GHC package. The call

```
Prelude> repeat 'a'
"aaaaaaaaaaaa...
```

will flood your display with a never ending sequence of 'a's. Admittedly, that is still easy to achieve with an imperative language emitting characters in a loop. Now transform the whole sequence by turning all characters to upper case.

```
Prelude> map Char.toUpper (repeat 'a')
"AAAAAAAAAAAA...
```

This is also possible in an imperative language, e.g. using iterator objects, yet more difficult. Finally we can clip the infinite list to a finite prefix.

```
Prelude> take 20 (map Char.toUpper (repeat 'a'))
"AAAAAAAAAAAAAAAAAAAA"
```

That is, we can formally work with an infinite amount of data. If only a finite amount of data is requested, which in turn does only depend on a finite amount of input data, then a result can be computed.

Another feature of the functional paradigm are *higher order functions*. These are functions that take other functions as input or return functions as output. While the first one is possible in most popular imperative languages, the second one is usually impossible. However taking functions as input is much more used in functional languages than in imperative ones. It gives you the combinatorial power of combining functions that traverse through a data structure and functions that process atomic data. It allows the programmer to create custom loop structures, which take the loop body and termination criteria as parameters of function type. In short, higher order functions allow definitions of loops, conditional branching and exception handling with the core language elements. Thus, these control structures need not to be hard-wired into the language.

Also returning functions as output is very common. In fact most **Haskell** library functions do that. E.g. instead of defining the concatenation of two lists `(++)` as a function from a pairs of lists to a list, which would mean type `([a],[a]) -> [a]`, the operator has type `[a] -> ([a] -> [a])`. (The parentheses could be omitted, since `->` is right associative.) Converting the first form to the second one is referred to as *currying*, in honor of HASKELL BROOKS CURRY, who lent his first name to the programming language we discuss here. However, it should be noted once again, that the technique is due to SCHÖNFINKEL, whose name did not seem to be appropriate for a verb like SCHÖNFINKELN (German) or SCHOENFINKELING (English). [Tho99] With the curried type of ++ it is possible to give only the first argument, which results in a function. This is called *partial application*. The expression `([1,2,3] ++)` denotes the function which prepends `[1,2,3]` to a list. Try yourself!

```
Prelude> :type ([1,2,3]++)
([1,2,3]++) :: (Num a) => [a] -> [a]

Prelude> ([1,2,3]++) [4,5,6]
[1,2,3,4,5,6]
```

To be precise, `([1,2,3] ++)` is a special syntax for partial application on infix operators, called *sectioning*. It is also allowed to write `(++ [1,2,3])` for the function that appends `[1,2,3]` to a list. Since strings are lists of characters in **Haskell**, the following is possible.

```
Prelude> map ("<"++) (map (++">") ["html", "head", "body"])
["<html>","<head>","<body>"]
```

Higher order functions and partial application allow for a very compact notation which reduces the need for specialized library functions or library functions that do very different things depending on type and number of the passed arguments. Even more, the latter is not possible in **Haskell** because it would conflict with **Haskell**'s static typing and the partial application of functions.

3 Architecture

In this section we describe the components of the parallel-web translation tool.

3.1 Data structures

First let us introduce some types, that are essential for our work. We do not give the exact types of the implementation but slightly simplified ones.

The first type is `Name`, which is used for tag and attribute names. It will use lower case letters for simple processing. The simplest definition for `Name` would be a *type synonym* for `String`, that is, `Name` would be just a textual substitute for `String`.

```
> type Name = String
```

Actually, in the implementation it is declared by `newtype` as a type which is incompatible to `String`, in order to increase type safety.

An attribute is a pair of a name and a value. The value is in principle a string, but we need different sorts of strings, as we will see later. Thus the type synonym `Attribute` has the *type parameter* `str`.

```
> type Attribute str = (Name, str)
```

In the first phase of the HTML document processing the input is converted to a list of tags. It need not to satisfy properties like matching open and close tags, and thus is also called a *tag soup*. The respective **Haskell** library is useful in itself for simple tasks that do not require a valid HTML structure, like collecting all papers that an author presents on his publication list in the WWW. But this library also proved useful as a lexer for a HTML tree parser.

For a tag a type synonym is not powerful enough. We need an *algebraic data type* (???) (for C programmers: a union). This is declared by `data` and such types are distinct from all other types defined this way. The following definition says that a tag is either an open tag with a name and a list of attributes, a close tag with a name, a plain text, a comment, a special tag like `<!DOCTYPE>` or a warning. Warnings are inserted in the tag soup by the lexer whenever syntax errors occur. Ampersands that are not part of a HTML entity references are such errors.

```
> data Tag str =
>      TagOpen Name [Attribute str]
>    | TagClose Name
>    | TagText str
>    | TagComment String
>    | TagSpecial String String
>    | TagWarning String
```

Suitable choices for the `str` parameter are the built-in `String` type based on Unicode characters and other string types provided by libraries that address for instance efficiency. For HTML/XML processing we also need a type which handles XML entities. An XML character can be a plain character, a numeric character reference or a character entity reference.

```
> type XMLString = [XMLChar]
>
> data XMLChar =
>      Char Char
>    | NumericRef Int
>    | NamedRef String
```

For our application a tag soup has not enough structure. We need to distinguish text enclosed in `<script>` tags and tags in a `<form>` environment from data in other contexts. To this end we define a very generic tree structure. A tree is recursively defined by being either a branch with branch specific data and a list of sub-branches, or a leaf with leaf specific data.

```
> data Tree branch leaf =
>      Branch branch [Tree branch leaf]
>    | Leaf leaf
>   deriving Show
```

For this generic tree we can already define traversal functions that we can later use for our HTML tree. A simple one, say `mapLeaves`, may convert all leaf information using a function of type `a -> b`.

```
> mapLeaves :: (a -> b) -> Tree branch a -> Tree branch b
> mapLeaves f t =
>    case t of
>       Branch b ts -> Branch b (map (mapLeaves f) ts)
>       Leaf   l    -> Leaf (f l)
```

Once again we want to stress, that lazy evaluation allows us to work with trees as if they are in memory completely. But actually they are only build as deep as needed and they are flushed if obviously no longer needed. However, see section 4.2 to see how we must take care, that there is no need to keep the whole HTML tree in memory.

We define a HTML tree as a generic tree equipped with more concrete types for branch and leaf information. More precisely, tags are the branches and texts, comments, warnings are the leaves. The string type is again a parameter. Since the HTML data structure can be used more generally for XML processing and since the famous Haskell libraries for HTML processing, namely HaXML and HXT, also address primarily XML, we call the type for a HTML tree `XMLTree`, too.

```
> type XMLTree str = Tree (Branch str) (Leaf str)
>
> data Branch str =
>      Tag Name [Attribute str]
>
> data Leaf str =
>      String  str
>    | Comment String
>    | Warning String
```

3.2 Rewrite hyper links

Updating hyper links in a document such that referred documents are translated, too, is the feature, that distinguishes the parallel web from other online translators.

There are four kinds of links. For the following assume that we translate a document from `http://original.net/` and the translator on the translating server is called `translator`.

1. Links to fragments in the same document like `#section2` need no change.
2. Links to binary data like in `<img src="...">` and `<body background="...">`, must be converted to absolute addresses, since the relative ones refer to the translating server. In our example setting, the relative link `content.html` would be converted to `http://original.net/content.html`.
3. Links to HTML documents or plain texts like in `<a href="...">`, `<meta http-equiv="refresh" content="...">`, `<link rel="contents" href="...">` must be converted to absolute URLs and equipped with the URL of the translator. The URL of the translator can be a relative one, since from the perspective of the web client the translating server originates the translated page. The link `content.html` would be replaced by `translator?source=http%3A%2F%2Foriginal-server.net%2Fcontent.html`.

   There is an exception from the rule: Hyper links may point to binary files. To be sure of what type the linked content is, we would have to request the HTTP headers for all referred documents, which is obviously too much overhead. Instead we inspect the file name extensions and guess whether their content is binary or text. This heuristics works rather reliably in practice.
4. HTML forms require more effort. Firstly there is the `action` attribute of the `<form>` tag which is extended by the translator's URL as explained above. Then there are the parameters of the translator and the parameters of the form that must be mixed in the same form. To this end all parameters of the translator are inserted as `<input type=hidden name="name" value="value">` into the `<form>` container and the names of all `<input>` tags of the original form become prefixed with `"orig"`. If the translator is called by submitting a form, then it reverses that procedure and calls the original server with accordingly prepared CGI parameters.

   This method is the same both for GET and POST requests. The difference is entirely in the reception of the CGI arguments.

3.3 Translation

The translation phase is divided into two steps.

1. Identify the texts that can be translated.

2. Translate the translatable text.

We could also write a function which traverses the HTML tree and applies a given function to translatable text. However we will see that in **Haskell** there is an important difference between stateless and stateful translators, which require different traversing procedures. Separating the identification of translatable text from the actual translation avoids code duplication.

Typical examples of translatable text are plain text, alternative image descriptions in `<img alt="...">`, and button texts in `<input type=submit value="...">`, unless their content is submitted to the CGI server (which is the case if the `name` attribute is also present). Typical examples of non-translatable text are plain text enclosed in `<script>` and most attribute values, including those containing hyper links.

We have a general problem with stylistic markup in a text: If only a part of a word is highlighted, say `<b>ir</b>regular`, the word appears as multiple words to the translator. However, it is generally not possible to highlight the translated word equivalently.

For word-wise translators the text is additionally split into words and punctuation, words into sub-words, and words are canonically turned to lower-case.

```
*Grammar> splitText "MetaFont, MetaPost and PostScript"
[(CapitalizedWord,"meta"),(CapitalizedWord,"font"),
(Punctuation,","),(CapitalizedWord,"meta"),(CapitalizedWord,"post"),
(Punctuation," "),(LowerCaseWord,"and"),(Punctuation," "),
(CapitalizedWord,"post"),(CapitalizedWord,"script")]
```

As mentioned above, the translators fall into two categories:

- Stateless translators translate cannot remember anything of what they have translated so far. Every occurrence of the same word is translated to the same translated word. The order of the input words determines the order of the output words, but not their translation. A simple example is the replacement of all vowels by the same vowel. The type of a stateless translator is `String -> String`.

- Stateful translators keep track of a state. Reordering the words can change the result to something entirely different. Examples are translators that act pseudo-randomly (their state is the random seed), and a translator which adds word counts to the words (its state is the counter).

  The type of a stateful translator is `String -> (s -> (String, s))`. That is, given a text and a state of type `s`, the translated string and the updated state are returned. This is usually condensed to `String -> State s String` and further generalized to `Monad m => String -> m String`, where, as the type suggests, `m` is a *monad*. Monads are the usual way to handle state in **Haskell**, but the concept is more general and by far not restricted to stateful computation. Monadic tree traversal functions immediately allow for non-deterministic translators, translators with I/O interaction, or with exception handling.

  For a nice introduction to monads for I/O interaction see [PJ02].

What would have happened, if we translated texts immediately as we identify it? The problem is, that in contrast to imperative programming style we want to separate the logical steps. We want to say "translate all texts of kind A in the tree and translate all texts of kind B in the tree". Consider the tag `<tag attrB="first" attrA="second">` where `attrA` belongs to kind A and `attrB` to kind B. If we merge the identification and translation, then the word counting translator results in `<tag attrB="first(2)" attrA="second(1)">`. This is not only counterintuitive (note the pun!), but it does also obstruct lazy processing: `attrB` can be processed only after `attrA` has been read. Thus its content must be kept in memory for the time `attrA` is translated.

Stateful and stateless translators could be unified as stateful translators, where originally stateless translators get an empty state. Would it be a good idea to handle stateful and stateless translators with the same traversing function? Certainly not. Stateless translators are much more flexible in use. Since they do not depend on each other, the compiler can freely choose the order of execution, it could in principle distribute translation of several parts over multiple processors. A very practical aspect is efficiency: If you request only one paragraph of a translated document, **Haskell**'s run-time system will parse the input document only until the end of the corresponding input, but actually translates only the requested paragraph. In contrast to that, a stateful translator depends on all translations done before, that is, for translation of the requested paragraph the whole text preceding it must be translated, too.

We observe two things:

1. Due to **Haskell**'s pure functional nature and its strict type system, dependencies of a function on a state must be made explicit by its type and due to laziness we must take care, not to make everything stateful.

2. However we can use interim data, to separate out all steps that are not affected by state. Laziness allows to combine them again with almost no loss of efficiency. A compiler optimization called *fusion* will probably remove these interim data structures.

That is, laziness is allowed only by the pure functional paradigm, but laziness weakens the inconveniences introduced by the pure functional paradigm.

3.4 Other components

The parallel web is not a monolithic project but uses several other **Haskell** libraries.

- The **Haskell** Web Server [Mar00] is used for delivering translated pages. Extended versions with CGI scripts exist [Thi02, Bri07], but this basic version was chosen for reasons of security.

- The HTTP library [GB+07] is used as HTTP client for downloading HTML pages from other servers.

- From HaXML [Wal07] and HXT [Sch07] some data structures, data tables and functions are imported, like those for decoding characters (UTF-8, ISO-Latin) and XML entities and relations of HTML tags (self-closing tags like `<br>`, mutual closing tags like `<p>`, `<li>`, `<tr>`).

- The Tag Soup library [Mit07] was extended and used as a lexer for the HTML tree parser.

3.5 Putting everything together

For our translation application we employ lazy evaluation as follows: We want to keep as little as necessary data in memory. That is, we want to read data from the source, translate it and quickly send it to the client. For the translation of current data it is not necessary to know whether more text follows, may it be infinitely much or may there be a read error. In an imperative language this would be solved by a loop, that repeatedly reads a reasonable amount of input, translates it, sends it to the web client. In contrast to that, we do not want to program and test these atomic steps, but we want to separate our program into logical steps. Each of this logical steps transforms formally the whole HTML document, but when these steps are applied in sequence, then actually they are applied simultaneously. That is the document is scanned only once.

The imperative approach makes it hard to parameterize the steps within the loop. Say you want to skip the hyper link modification on user request. You have to extend the loop to check this condition. Differently with lazy evaluation: We can omit the link modification step conditionally in the chain of transformations. Or consider the conditional translation of parts of the document. E.g. program text should not be translated by a translator that is designed for natural languages. Thus we want to say: Text enclosed in `<script>` and `<pre>` tags shall not be translated. In our implementation we write it just this way. In an imperative loop based implementation we had to maintain a loop-global variable which says whether to translate, and whose value is switched whenever `<script>` or `</script>` occurs.

The list of logical steps follows. Each step corresponds to a function, which transforms formally the whole input at once. Actually, when applied in sequence, the atomic operations are executed in an interleaved way as the generation of the output requires it. E.g. for computing the first output character the program finds out, that the first tag must be known. In order to know the first tag a sufficient amount of characters must be read from the input and so on.

The logical transformations are

- Fetch data from foreign server (HTTP.Get)
  `IO String`
- Lex tags
  `String -> TagSoup XMLString`
- Parse into HTML tree
  `TagSoup XMLString -> XMLTree XMLString`
- Determine encoding from `<meta http-equiv="Content-Type">` tag
  `XMLTree XMLString -> Encoding`
- Decode characters from UTF-8, ISO-Latin-1 etc. to Unicode
  `Encoding -> XMLTree XMLString -> XMLTree XMLString`
- Decode HTML entities to Unicode characters
  `XMLTree XMLString -> XMLTree String`
- Modify links
  `XMLTree String -> XMLTree String`
- Identify translatable text
  `XMLTree String -> XMLTree (Bool, String)`
- Translate non-monadic
  `(String -> String) -> XMLTree (Bool, String) -> XMLTree String`
- or translate monadic
  `Monad m => (String -> m String) ->`
  `XMLTree (Bool, String) -> m (XMLTree String)`
- Encode Unicode by HTML entities
  `XMLTree String -> XMLTree XMLString`
- Format HTML
  `XMLTree XMLString -> String.`

4 Technical issues

4.1 Strong and static typing

Strong typing is suspected for being inefficient in the world of machine oriented programming dominated by C. Static typing is mistrusted by the world of scripting languages represented by PHP, Perl, Python, Ruby, Rebol, Rexx, Lua, because of expected lack of flexibility. **Haskell** is both strong and statically typed, yet compilable and at a very high level of abstraction. Several experimental type extensions to the standard **Haskell** 98 indicate the wish for more flexibility, while not giving up type safety entirely. However, the largest part of the parallel web project does not need any extension. Here we want to sketch one typical occasion where strong static typing proved being very useful.

It is seldom that programs are designed the right way from the beginning. When it comes to the details, problems pop up, that are optimally solved by some global restructuring. Early versions of the parallel web used the `XMLString` as the only data type. That is, the earlier `XMLTree` meant, what is now represented by `XMLTree XMLString`. `XMLString`s were converted to Unicode `String`s and back for each translation. It would have become an efficiency problem, if multiple translations would be applied. But it was also immediately a problem, since the conversion from and to XMLString also contained the conversion of special characters to HTML entities. That is, both ä and the ISO-Latin ä were converted to ä. This saved us encoding procedures for various character encodings. The problem was that non-translated texts did not passed this canonicalization, and thus it happened that a UTF-8 encoded character slipped through to a document without a character set specification.

This and the efficiency problem of duplicate conversion were solved by parameterizing the XMLTree with a string type parameter. Now in a first step all strings are converted from XML to Unicode, then several translations can take place, then the Unicode strings are translated back to a canonical XML string. Static typing protects you from accidentally placing a translator in the wrong phase, i.e. translating XML strings by a translator based on Unicode.

The rewriting consisted of adding a type parameter to each function signature and checking how general the function is (XMLString only, String only, any string type). This was tedious, but the compiler catched all missing or wrong replacements and in the end the code worked immediately, again.

4.2 How to keep lazy

Once you have implemented the logical steps with a maximum of laziness, i.e. with a minimum of data dependencies, it is simple to compose the steps in an interleaved manner. However, lazy implementation requires care. E.g. if you want to strip all spaces at the end of a string it is convenient to reverse the string, strip the leading spaces and reverse the character order again.

```
Prelude> reverse (dropWhile Char.isSpace (reverse "test text    "))
"test text"
```

However the first character of a reversed string is the last character of the input string. Thus requesting only the first character of the reversed string means consuming the whole input string. A solution which preserves laziness must process the list beginning from the end. An implementation of this idea is not longer than that of dropWhile, but it is certainly not obvious, why it reduces data dependencies.

```
Prelude> let dropWhileRev p = foldr (\x xs -> if p x && null xs then [] else x:xs) []
Prelude> dropWhileRev Char.isSpace "test text    "
"test text"
```

Another example is the handling of errors. The HTML parser could be designed to return an error message instead of a HTML tree in case of a HTML syntax error. We would give it the type String -> Either String XMLTree. However, for the decision whether there is an error in the whole document the complete input must be parsed. This design clearly forbids a lazy transformation of data. Thus we must handle errors as they arise. We could collect them in a list (type String -> (XMLTree, [String])), or we store them in the XMLTree structure. As mentioned in section 3.1, we used the second approach and emit errors as comments after translation.

Character encoding in HTML confronts us with a paradoxical situation: The encoding can be given in a <meta> tag which itself is encoded. This only works under the assumption that the content of the <head> tag can be parsed successfully as ASCII (or ISO-Latin-1) text. We must scan for such a <meta> tag, but we must be prepared that no such tag is present. We can start decoding only after the decoding information is found or we become confident that there is no such information in the document. It is essential that we stop the search at the end of the <head> tag, or at the beginning of <body>. This way, the first character of the output depends on all characters of the <head> block in the input, but there seems to be no way to reduce this dependency.

In summary, the programmer has to care about keeping the data dependencies at a minimum.

4.3 Security

On the one hand a **Haskell** web server perfectly fits the "security by obscurity" philosophy, buffer overruns are in principle not possible, and type safety reduces unexpected situations at run-time. On the other hand the **Haskell** web server still lacks security techniques, like HTTPS and SSL, that are found in the big standard servers. Additionally the increasing popularity of **Haskell** together with provocative claims like "**Haskell** is safe" may encourage people to search and finally find security holes also in a **Haskell** web server.

5 Conclusion

Haskell cannot do magic, but makes possible things maintainable. You still have to program, it needs time, and you can make errors. However errors can be found early, often at compile time.

The development of the parallel web takes place at SourceForge `http://sourceforge.net/projects/parallelweb` and a running version of the server can be tested at `http://www.parallelnetz.de/`.

References

[Bri07] Björn Bringert. Haskell Web Server 2: A web server including CGI. `http://www.cs.chalmers.se/~bringert/darcs/hws-cgi/`, 2007.

[Dau04] Hal Daume. Yet Another Haskell Tutorial. `http://www.cs.utah.edu/~hal/docs/daume02yaht.pdf`, 2004.

[GB+07] Warrick Gray, Björn Bringert, et al. Haskell HTTP package. `http://www.haskell.org/http/`, 2007.

[HHPPJ04] Kevin Hammond, Cordelia Hall, Will Partain, and Simon Peyton Jones. GHC: The Glasgow Haskell Compiler. `http://www.haskell.org/ghc/`, 2004.

[Hud89] Paul Hudak. Conception, evolution, and application of functional programming languages. *ACM Comput. Surv.*, 21(3):359–411, 1989.

[Hug89] John Hughes. Why functional programming matters. *The Computer Journal*, 32(2):98–107, 1989.

[JR04] Mark P Jones and Alastair Reid. Hugs 98. `http://www.haskell.org/hugs/`, 2004.

[Mar00] Simon Marlow. Writing High Performance Server Applicatins in Haskell. Case study: A Haskell Web Server. Technical report, Microsoft Research Ltd., Cambridge, July 2000.

[Mit07] Neil Mitchell. Haskell Tag Soup library. `http://www-users.cs.york.ac.uk/~ndm/tagsoup/`, 2007.

[PJ98] Simon Peyton Jones. Haskell 98 language and libraries, the revised report. `http://www.haskell.org/definition/`, 1998.

[PJ02] Simon Peyton Jones. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in haskell. Technical report, Microsoft Research, Cambridge, July 2002.

[Sch07] Uwe Schmidt. Haskell XML Toolbox. `http://www.fh-wedel.de/~si/HXmlToolbox/`, 2007.

[Thi02] Peter Thiemann. WASH/CGI: Server-side Web Scripting with Sessions and Typed, Compositional Forms. Technical report, Universität Freiburg, 2002.

[Tho99] Simon Thompson. *Haskell: The Craft of Functional Programming*. Pearson, Addison Wesley, second edition, 1999.

[Wal07] Malcolm Wallace. HaXml. `http://www.cs.york.ac.uk/fp/HaXml/`, 2007.