Adapting Web Contents to WAP Devices using Haskell

Pedro Ivo Oliveira and Carlos Camarão
DCC, UFMG, Brazil
{camarao, ivo}@dcc.ufmg.br

August 21, 2001

Abstract

This work describes an implementation of a system for adapting the contents of HTML Web pages so that they can be shown adequately in mobile devices with WML microbrowsers. The article reports on the experience of using a lazy functional programming language and, in particular, compilation by composition of monadic combinators, in the development of a practical application in a modern and emerging technological area.

1 Introduction

The Internet is a rich source of information, which can be used freely, at any instant of time. A significant limiting factor for further spreading of access and use of this information, among others such as the data transfer rate, is that users must still make use of a connection that is physically attached to a computer.

With dropping in prices of integrated circuits and the evolution of technology of wireless devices and their interconnection[21], we have recently seen the development of new forms of overcoming this limitation, from the creation and development of *notebooks* and portable computers, through personal digital assistants and cellular phones connected to portable computers, to a new era of mobile phones which can access information available in the Internet directly. New data transmission protocols have been created, which take into account the limitations of mobile phones for access to data in the Web (see Figure 1).

A large part of devices allowing access to the Web use the WAP protocol[20, 4]. This protocol is more adequate to data transmission in wireless applications, because it requires fewer control and data processing than TCP/IP[24, 3] (the conventional protocol used for data transmission in the Internet).

Documents in the Web are created by using *mark-up languages*, characterized by the use of marks to specify document structure and layout. The mark-up language most widely used in the Web is HTML[25]. However, WAP devices use WML[20, 5]. The amount of information available nowadays in the Internet in the form of WML documents is quite minute, in comparison to the enormous amount of information available — and, moreover, frequently modified — in the form of HTML pages (in the order of 1 billion pages).

Figure 1: Limitations of mobile devices for Web navigation

There exists, therefore, a strong and clear motivation for experimentation with tools and systems for translating HTML to WML, and in general for *adapting* Web pages so that they can be shown in a suitable form in wireless devices. Figure 2 illustrates the situation of a WAP device with a WML microbrowser making accesses to Web pages in HTML by using a translator from HTML to WML. In this paper we report on an experience of employing a recently developed compilation technique in the development of a practical application in this modern, emerging technological field, with potentially highly influential consequences in software and high-tech industry.

This technique is based on the composition of monadic combinators[12, 13, 9, 23], explored for example by the language Haskell[6, 9]. The next section

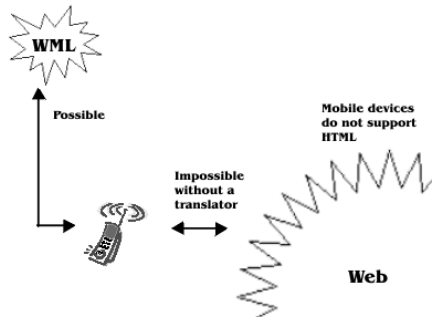


Figure 2: Accessing data from mobile devices

provides a description of the main characteristics of monadic parsing/compilation, as well as a comparison with a more traditional approach based on the use of a parser generator system like, for example, YACC[19]. We consider the monadic technique a significant landmark in the field of compiler development, which has not yet been duly explored in practice, in particular in cases in which the efficiency of generated compilers are not of utmost importance.

2 General Description

The system developed for adaptation of Web contents to WAP devices consists of a *Text Adapter*, a *Personalized Adapter* and a *Page Splitter*.

The text adapter takes HTML texts[25] extracted from Web pages and adapts them in order to generate WML decks and cards that can be shown in the screen of a mobile device. An example is shown in Figure 3.

The major difficulties in the implementation of this tool were due to the facts that, firstly, the HTML language has a great number of elements, with corresponding attributes, and their use involves recognizing a lot of small rules and details; secondly, a significant number of these rules introduce context-sensitive conditions, some of which introduced because of irregularities contained in most Web pages throughout the Internet, and allowed by HTML browsers commonly used. These irregularities do not conform to the W3C specification[25].

As an example of a simple adaptation performed by the text adapter, consider the translation of the following HTML element:

```
<img src="http://www.todowap.com.br/figs/celular.jpg">
```

The text adapter translates Web pages into plain text and, when it sees, for example, a figure, it simply indicates that a figure with a certain name existed in that

Figure 3: HTML and corresponding WML

place. In this case, it simply extracts the name of the figure and introduces it, between square brackets, in the current position of the WML card being generated, producing for the HTML element above simply:

```
[celular.jpg]
```

The Personalized Adapter, on the other hand, allows the user to specify figures, links and alternative texts to be included in a WML document. It treats special marks inserted in an HTML document, with the aim of generating a better translation. In the absence of marks, it behaves just like the Text Adapter.

The translation rules create simple WML code (based on WML 1.0[5]), allowing generated documents to be visualized in any WAP device.

The Page Splitter divides a WML document, if necessary, into smaller documents that can be sent to a particular WAP device. The splitting is done when the size of the memory necessary for storing a document is greater than the WAP device's cache memory.

Section 4 provides more details of the functionality of these tools.

3 Implementation Overview

The implementation is essentially a syntax-directed recursive descent monadic translation, as described in the next subsection. Monadic translators are defined by

composing previously defined ones by means of combinators, starting from basic character and string translators. We call a monadic parser simply a monadic translator that just parses the input, not producing any useful result (translation).

We comment next on the fundamental advantages and disadvantages of this approach, in comparison to a more traditional one, of using a parser generator.

The most important advantage of monadic translation comes from the possibility of using the abstractions and mechanisms available in Haskell, for writing new monadic translators and combinators, in a clean and compositional way. This occurs in contrast with the source code of a compiler generated automatically, with semantic rules written in C. Section 3.1 highlights the advantages of using polymorphic higher-order functions. A second and related advantage comes from avoiding to deal with the problems of errors and grammar conflicts associated with using a LALR(1)-based table-driven parser generator (like, for example, YACC).

On the other hand, we do not have yet a monadic combinator library that is fast enough to compete, in terms of runtime efficiency, with parsers generated automatically from LALR grammar descriptions (but see Section 5). The interested reader can consult the works of Swierstra and others [23, 11, 18] in this respect. Another advantage of using automatically generated parsers from an LALR(1) grammar description is that ambiguity in the specification of grammar rules is automatically detected by parser generators. However, this has not represented a big problem in our case, because the HTML language does not introduce cases of ambiguity.

3.1 Monadic Translation

Monads were originally used as a way of structuring the semantic descriptions of computational features such as state, exceptions and continuations[22]. Later, they were used as a technique for structuring functional programs and representing state changes[30, 28]. As part of these works, a parser was defined and used as an instance of a monad[12, 13, 9]. This has renewed interest in this area and conveyed a more profound understanding of the behaviour of parsers. This was to a certain extent surprising, given the fact that this is an area that had already received a great deal of attention by many researchers.

The type of a translator can be defined in Haskell as follows:

```
newtype Translator a =
  Translator (String -> [(a, String)])
```

This means that a translator receives a string as input and returns a list of pairs: the first component of each pair represents a possible translation obtained as the result of parsing and consuming part of the input, and the second component represents the unconsumed part of the input, after parsing.[1]

For example, parser `item` defined below consumes and returns the first character of an input string, always with success:

```
item::  Translator Char
item = Translator (\s0 -> case s0 of
           ""    -> [];
           (c:s) -> [(c,s)])
```

The first component of the result, `c`, is the translation of the consumed character (itself `c`) of the input `s0`, and `s` is the rest (the unconsumed part) of the input, after parsing.[2] As will become clear in the sequel, a "failure" is represented by an "empty list of successes"[27].

The monadic combinators `>>=` (called `bind`) and `return` define respectively a translator which returns a value without consuming any part of the input, and a translator that combines two translators sequentially, passing the result of the first to the second:[3]

```
instance Monad Translator where
  return a = Translator (\s -> [(a, s)])
  p >>= f  = Translator (\s -> concat
    [app (f a) s' | (a, s') <- app p s])
```

[1] Symbol `->` is Haskell's notation for specifying a functional type (`A -> B` is the type of functions from `A` to `B`). Symbol `[]` is the list type constructor (`[A]` is the type of lists of elements of type `A`) and `:` and `nil` are list value constructors (`nil` denotes the empty list and `1:nil` the list whose first element (or head) is `1`, of type `Int`, and the tail is the empty list). The notation `[1,2]` is an abbreviation of `1:(2:nil)` (representing the list formed by the elements `1` and `2`, of type `Int`). `(,)` is the (type and value) constructor for pairs (`(A,B)` is the type of values whose first component has type `A` and the second type `B`, and `(1,'2')` is a pair whose first component is `1`, of type `Int`, and the second is `'2'`, of type `Char`).

[2] `\x -> e` is Haskell's notation for the anonymous function with the same behaviour of function f given by `f x = e`. `"` is the delimiter for string literals (`""` is the empty string).

[3] `[f(x) | x <- l]` is a list comprehension, and can be interpreted analogously to a set comprehension, commonly used in mathematics; it represents the list of elements `f(x)`, for all elements `x` of list `l`. Function `concat` concatenates a list of lists to form a single list.

where `app` is simply the deconstructor function of datatype `Translator`:

```
app (Translator p) = p
```

An empty list represents a failure, in the sense that, if `app p s` returns an empty list, then `app (p >>= f) s` also does (due to the behaviour of list comprehensions), for *any* well-typed f. Thus, after such a failure, no further processing is done on the input, which means that parsing stops after such a failure. This is not specific to this monad, but occurs for each monadic type with a zero[29].

Now, as the second translator, `f a`, can depend on the result of the first, monadic translation is able to receive as input languages that can only be described by context-sensitive grammars, and are neither LL(k) nor LR(k), for any k. This is not surprising. It is often necessary, when using a parser generator (like YACC, for example) to have to resort to semantic rules for input validation. A simple example, which happens to occur quite often, of a language rule that is context-sensitive, can be referred to as "use-as-defined": the uses of a variable (or any other defined entity) should be such that it conforms to its definition.

Parsing of HTML elements is an example where this is useful, since the context determines, in many cases, whether an occurrence of a given element is valid or not, and in the first case what meaning it conveys to the structure or layout of a document.

Using the monadic operations of `return` and `>>=`, we can define new translators, in a simple and compositional way. Haskell's `do` notation[15] is helpful, albeit a construct that can be defined in terms of the basic monadic combinators: `do { x <- p; q}` is used frequently, meaning `(p >>= (\x -> q))`, due to its natural operational reading: "first apply `p`, binding its result to `x`, and then apply `q`.

Consider now the example of the higher-order function `sat`, that receives as parameter a predicate and returns a translator that just consumes a single input character if this character satisfies the predicate, and otherwise fails:

```
sat::  (Char -> Bool) -> Translator Char
sat p = do item
           if p x then return x else zero
```

where `zero` denotes a translator that always fails, for any input:

```
zero::  Translator a
zero = Translator (\s -> [])
```

Using `sat`, lexical parsers that parse an specific character, a lowercase letter and an uppercase letter can be defined very simply:

```
char c = sat (\c' -> c==c')
lower  = sat isLower
upper  = sat isUpper
```

Here `isLower` and `isUpper` are predicates defined in the Haskell `Prelude`.

The monadic combinator `orelse`, defined below, is used often in our translators. It implements a deterministic choice: `(p `orelse` q)` behaves like `p` unless `p` fails, in which case it behaves like `q`:[4]

```
p `orelse` q = Translator (\s ->
  let ps = parse p s
  in if null ps then parse q s else ps)
```

Combinators `some` and `many` provides other examples of frequently used ways of combining translators: when applied to an input string, `some p` applies `p` repeatedly to this input, *one* or more times, inserting the result of each such application into a list, whereas `many p` applies `p` *zero* or more times:

```
some::  Translator a -> Translator [a]
some p = do a <- p
            as <- many p
            return (a: as)

many::  Translator a -> Translator [a]
many p = some p `orelse` return []
```

Note that `many p` tries to use `p` and, if `p` fails, in its first application, the input is left intact.

As a simple examples, we can define a parser for a string of spaces, tabs or newlines, as follows:

```
space::  Translator String
space = many (sat isSpace)
```

[4] Dyadic functions may be used in infix form, like operators, in Haskell, by simply using backquotes before and after the function's name, as in `p `orelse` q`.

Continuing in this way, we are able to build other needed translators and combinators. Our final example illustrates the definitions of i) a parser combinator `token`, which receives a parser `p` and returns a parser that behaves like `p` but throws away trailing spaces; ii) a parser `string` that parses a given string (and fails if the sequence of characters from the input does not match the given string); iii) a parser `symb` that composes `token` and `string`; iv) part of a interpreter (for expressions) that treats `let` expressions, assuming that `var` is a parser for identifiers (variable names) and `exp1` treats other syntactic alternatives of expressions.

```
token::  Translator a -> Translator a
token p = do a <- p; space; return a

string::  String -> Translator String
string "" = return ""
string (c:s) = do char c
                  string s
                  return (c:s)

symb::  String -> Translator String
symb s = token (string s)

exp0 = do symb "let"
          v <- token var
          symb "="
          e <- token exp0
          symb "in"
          e' <- token exp0
          return (\env-> e'((v,e env):env))
       `orelse` exp1
```

The result of interpreting `let v = e in e'` is expressed clearly, as a function that receives an environment (`env`) and returns the result of the evaluation of `e'` in an environment in which `env` is modified, by associating `v` with the result of interpreting `e` in `env`.

The monadic type that represents translators, as well as the basic monadic operations `return` and `>>=`, can be modified to deal with various features, such as, for example, error handling (typically to register error positions), in a way that is highly transparent to the rest of the program using the monadic operations (see e.g. [30]).

4 System Tools

This section describes the functionality of the Text and Personalized Adapter, and the WML Page Splitter.

4.1 Text Adapter

The major difficulties in the implementation of this tool were mentioned in Section 2. The Text Adapter implements a mapping, specified by carefully chosen translation rules, from each HTML element and its attributes to a corresponding WML element or sequence of elements and corresponding attributes. In some cases, when there is no possible way of constructing a correspondence for an HTML element, this HTML element is discarded.

The following are illustrative examples of simple translation rules:

Frameset: WAP devices do not provide support for showing several documents simultaneously, in separate parts of a screen. A menu with a link to each of the several distinct documents in the frameset is therefore created, so that the user can choose which of the several documents should be shown. For example, given the following HTML code:

```
<frameset rows="20%,60%,20%">
<frame src="www.test.com.br/pg1.html" name="page 1">
<frame src="www.test.com.br/pg2.html" name="page 2">
<frame src="www.test.com.br/pg3.html" name="page 3">
</frameset>
```

the Text Adapter generates the following WML code:

```
<a href="http://www.test.com.br/pg1.html">page 1</a>
<a href="http://www.test.com.br/pg2.html">page 2</a>
<a href="http://www.test.com.br/pg3.html">page 3</a>
```

Body: This element is simply translated to a pair of WML elements, as shown below:

```
HTML: <body> ...HTML...  </body>

WML: <card><p> ...WML...  </p></card>
```

Lists : As WML does not provide lists, HTML lists are adapted, as shown below:

```
HTML: <ul>
          <li> First item
          <li> Second item
      </ul>

WML: <br/>
     <br/>- First item <br/>
     <br/>- Second item <br/>
     <br/>
```

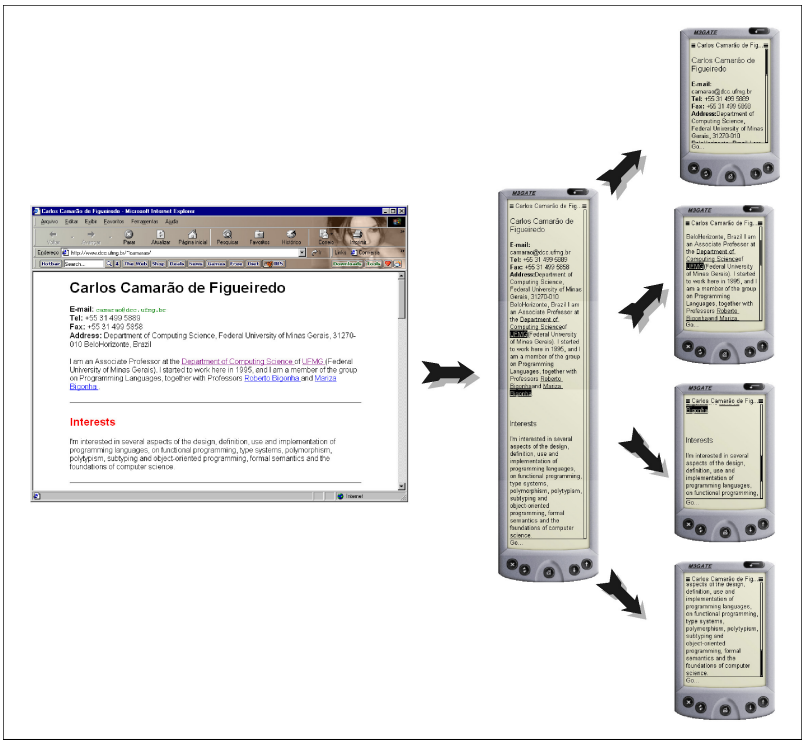


Figure 4: Fragmentation of WML pages

Character references: HTML allows the use of *character references* such as, for example, "`<`" (denoting <), and "`&quot;`" (denoting double quotes). There are thousands of character references in HTML, whereas WML 1.0 provides only 7.

Each reference to a special character is converted to its equivalent code in UNICODE[7]. For example, "` `" (a blank space) is converted to "` `".

Most translation rules are quite straightforward, but there are quite a great number of them, and a lot of details involving their use (as already mentioned in Section 2, due mainly to irregularities, with respect to the W3C specification, allowed by Web browsers). For example, many elements (like `block`) close a previously unclosed element (like `p`).

4.2 Personalized Adapter

The Personalized Adapter allows users to control the translation of HTML pages, in order to obtain results that are of better quality or are better suited for specific needs. To make this control, an HTML page developer may include certain marks for use by the adapter.

For example, a mark can be inserted to introduce an alternative title for the WML version of the page, so as to fit better in screens of mobile devices. Other marks can be used to show specific links or delimit parts of the text that should or should not be shown in the WML version. A page developer can also indicate figures with specific formats for WAP devices.

4.3 Page Splitter

The Page Splitter divides a WML document, if necessary, into smaller documents that can be sent to a particular WAP device (see Figure 4). The splitting is done when the size of the memory necessary for storing a document is greater than the WAP device's cache memory.

The general problem of dividing WML documents is rather complex, because of the existence of *atomic elements*, which cannot be separated and placed into two distinct documents. An example is the element describing a link:

```
<a href="http://www.todowap.com.br">Site WAP</a>
```

However, the pages generated by the adapter programs

are formed only by texts, links and images. Tables, lists and frames are adapted to a textual form, and forms and other complex structures are discarded. In this simpler framework, the fragmentation problem becomes manageable, since the Page Splitter uses only the following atomic elements: marks, words, elements `a` and `anchor` (used to specify a link), and `do` (inserted by some *gateways*[5]) to ease navigation in WAP pages.

5 Runtime Performance

As already mentioned in Section 3.1, parsers based on the use of monadic combinators are, nowadays, still slower than those automatically generated from LALR(1) bottom-up parser generators like YACC. For the adaptation of complex pages, the average rate of translation, measured in a Pentium II 450MHz Xeon, with 512 Mbytes of RAM, is approximately 15000 bytes/sec; for simpler (mostly textual) documents, the average rate is 39000 bytes/sec (with the translator compiled by using GHC's[16] optimization directive; without it, this rate is approximately a third of the above rates).

Current WAP devices have a wireless data transmission speed between 1200 and 1800 bytes/sec and maximum memory capacity of 3800 bytes (cf. description of second generation TDMA, CDMA and GSM in [1]). For devices of generation 2.5 (GPRS, see also [1]), this speed increases to between 8000 and 15000 bytes/sec, with memory capacity staying approximately the same as in devices of the second generation. Third generation devices have much higher speeds, and much more memory available, but they are not intended for operating with WAP protocols and the WML language.

In this context — i.e. considering the speed of WAP devices and the fact that they cannot store a large amount of data —, translation rates have proved to be quite satisfactory.

Runtimes of the Text Adapter, Page Splitter and both programs together are shown in Figure 5. Runtimes were measured on the machine described above, using a Perl script to repeat 1000 times, for each of the HTML pages used, the operations of the Text Adapter and Page Splitter, modified so as to read from and to generate a local file. We can divide the HTML pages used in two groups, the first containing images, figures and sometimes Java scripts, and the second consisting mainly of text. Pages `AOL`, `UOL`, `UAI`, `Estadão`, `IG`, `Matrix` and `Terra` are in the first group, and `Camarão`, `Haskell` and `Loureiro` in the second. The time taken by the fragmentation process is in practice not significant, since only a single card needs to be produced at first. The other cards may be subsequently generated while the user receives and interacts with previously generated ones.

The run times of both tools increase with the size of the input. More complex tags, like for example tags for images, take more time to be analyzed by the Text Adapter, in comparison to plain text and simpler tags. On the other hand, the Text Adapter tends to generate relatively less output code for complex pages, diminishing the Page Splitter runtime (in comparison to simpler pages).

6 Related Work

Related works consist mainly of commercial tools, and there are few or no technical information or performance data available. This makes it difficult to perform any detailed comparison. We comment on some of these works below.

The paper *VBXML: Converting HTML to WML*[10] presents a source code of a program that converts HTML to WML pages. It uses the approach of converting from HTML to XHTML first, and then to WML. However, the code is rather rudimentary, having a naive design and containing incorrections and sources of inefficiency.

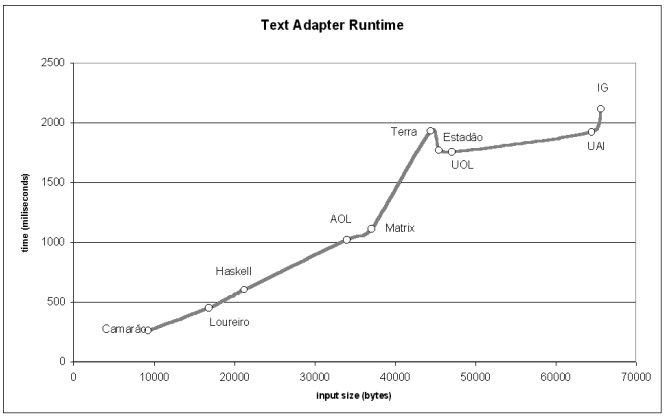
The commercial tools Zap2WAP[14] and WAPTool[8] make simulators available which show the result of adapting HTML pages to WAP devices. However, there are no technical information available about the translation process or the architecture of the system.

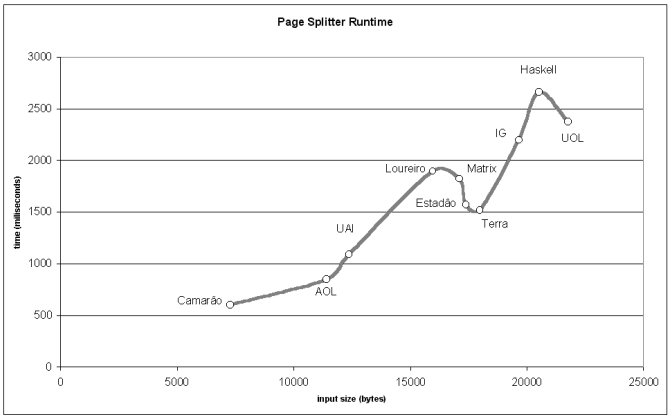
PyWeb[2] is a commercial tool that converts HTML to WML and is able of adapting the result according to the specific mobile device used. It also works with personalized marks that allow the user to specify how a document should be translated. Judging from the tests performed, which were run on a simulator, this has proven to be the best of all the tools we have analyzed. The approach used in the translation process is not described, neither other technical information about the system architecture.

The paper *Two Approaches to Bringing Internet Services to WAP Devices*[17] is another work similar to ours. The system architecture is described in the paper, but again there is no information about the approach used in the translation process. The related Web page does not present any significant additional information.

The approach for adaptation of HTML pages based on the use of XML — i.e. based on making two

[5] A gateway is a program responsible for the integration between the WAP network and the Web.





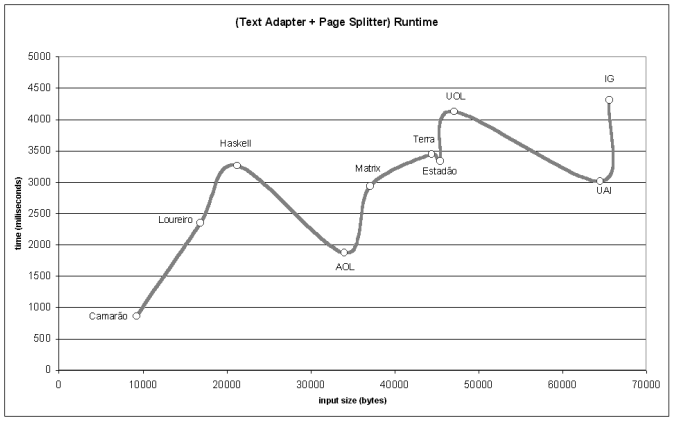


Figure 5: Runtimes of Text Adapter, Page Splitter and both tools together

conversions, from HTML to XHTML[26], and then from XHTML to WML, using XSL[31] — is often mentioned/cited, but we are not aware of any successful implementation currently available. This approach would also, of course, have the drawback of needing the development and use of two translators, and in the use of another, intermediary, mark-up language.

7 Conclusion

The Internet is a rich source of information, which can be used freely, at any instant of time. A significant limiting factor for further spreading of access and use of this information, among others such as the data transfer rate, is that users must still make use of a connection that is physically attached to a computer.

This work described programs that allow access to information available in the Web — in the form of HTML documents, the most widely used medium of data interchange in the Web —, in mobile devices with WML microbrowsers. These programs include a Text and a Personalized Adapter, as well as a WML Page Splitter. We have adopted a relatively unconventional approach for the development of these tools, namely, translation by composition and use of monadic combinators and translators. An architectural environment has also been defined and developed for the proper use of these tools.

The achievements and conclusions of this work can be summarized as follows:

- The adapters have demonstrated a quite satisfactory result for the visualization of HTML documents in mobile devices. It is encouraging to see that a significant amount of Web pages can be presented adequately in these devices.

- The use of special HTML marks for enhancing the quality of translated Web pages for presentation in WAP devices is quite important, to enable Web page developers to improve the quality of the presentation in cases where such an improvement is desired.

- The development of programs that deal with the whole of the HTML language requires a large amount of work, due to amount of details involved, and the use of Haskell abstraction facilities and monadic parsing/translation has enabled both a swift development of a working prototype as well as an easy modification by stepwise incremental extension of this prototype.

- The performance of the implementation has been quite satisfactory, given the speeds and the limited amount of data that can be stored in current mobile devices.

References

[1] Anywhere you go. http://www.ayg.com.

[2] PyWeb.com. http://www.pyweb.com/.

[3] TCP and UDP transport. www.medusa.uni-bremen.de/intern/orpc/node57.html.

[4] WAP Forum Specifications. www.wapforum.org/what/technical.htm.

[5] *Wireless Markup Language version 1.3.* www1.wapforum.org/tech/terms.asp?doc=WAP-191-WML-20000219-a.pdf.

[6] *Haskell, The Craft of Functional Programming.* Addison-Wesley, 1997.

[7] *The Unicode Standard, version 3.0.* Addison Wesley, 2000.

[8] Argogroup. Argogroup WAP Tool V1.1. http://www.argogroup.com/waptool/.

[9] R. Bird. *Introduction to Functional Programming using Haskell.* Prentice-Hall, 1998. 2nd ed.

[10] J. Britt. VBXML: Converting HTML to WML. www.vbxml.com/WAP/html2wap.asp, 2000.

[11] M. Chakravarty. Lazy Lexing is Fast. In *Fourth Fuji International Symposium on Functional and Logic Programming (LNCS 1722, Springer-Verlag)*, 1999. www.cse.unsw.edu.au/ chak/papers/papers.html.

[12] G. Hutton. Higher-order functions for parsing. *Journal of Functional Programming*, 2:232–343, 1992.

[13] G. Hutton and E. Meijer. Monadic parser combinators. Nottinham Univ. CS-TR-96-4, 1996.

[14] Jataayu Software. Jataayu's Zap2Wap. www.vsellinindia.com/jataayu/zapconvert_main.htm.

[15] S. P. Jones et al. The Haskell 98 Report. http://haskell.org/definition.

[16] S. P. Jones et al. GHC — The Glasgow Haskell Compiler. www.dcs.gla.ac.uk/fp/software/ghc/, 1998.

[17] E. Kaasinen et al. Two Approaches to Bringing Internet Services to WAP Devices. http://www9.org/w9cdrom/228/228.html.

[18] D. Leijen. Parsec combinator library. http://www.cs.ruu.nl/~daan/parsec.html.

[19] J. R. Levine, T. Mason, and D. Brown. *Lex & Yacc*. O'Reilly & Associates, 1992.

[20] S. Mann. *Programming Applications with the Wireless Application Protocol*. Wiley, 2000.

[21] G. R. Mateus and A. A. F. Loureiro. *Introdução à Computação Móvel*. Imprinta Gráfica e Editora Ltda., 1998.

[22] E. Moggi. Computational lambda-calculus and monads. In *IEEE Symposium on Logic in Computer Science*, 1989.

[23] S. Swierstra. Parser Combinators, from Toys to Tools. In *Haskell Workshop*, pages 113–128, 2000. http://www.cs.ruu.nl/people/doaitse/.

[24] A. S. Tanenbaum. *Computer Networks*. Prentice Hall PTR, 1996. terceira edição.

[25] W3C, http://www.w3.org/TR/html401. *HTML 4.01 Specification*, 1999.

[26] W3C. XHTML 1.0: The Extensible HyperText Markup. www.w3.org/TR/2000/REC-xhtml1-20000126/#toc, 2000.

[27] P. Wadler. How to replace a failure with a list of successes. In *Functional Programming Languages and Architecture, LNCS 201*, pages 113–128, 1985.

[28] P. Wadler. Comprehending Monads. *Mathematical Structures in Computer Science*, 2(4):461–493, 1990.

[29] P. Wadler. Monads for Functional Programming. *Computer and Systems Sciences*, 118, 1992. http://www.cs.bell-labs.com/who/wadler/topics/monads.html.

[30] P. Wadler. The essence of functional programming. In *Conference Record of the 15th ACM Symposium on Principles of Programming Languages*, 1992.

[31] Extensible Stylesheet Language (XSL). http://www.w3.org/Style/XSL, 2001.