Towards Responsive Retargeting of Existing Websites

Gilbert Louis Bernstein
Stanford University
gilbert@gilbertbernstein.com

Scott Klemmer
UC San Diego
srk@ucsd.edu

ABSTRACT
Websites need to be displayed on a panoply of different devices today, but most websites are designed with fixed widths only appropriate to browsers on workstation computers. We propose to programmatically rewrite websites into responsive formats capable of adapting to different device display sizes. To accomplish this goal, we cast retargeting as a cross-compilation problem. We decompose existing HTML pages into boxes (lexing), infer hierarchical structure between these boxes (parsing) and finally generate parameterized layouts from the hierarchical structure (code generation). This document describes preliminary work on ReMorph, a prototype 'retargeting as cross-compilation' system.

Author Keywords
Responsive Design; Document Layout; Webpages; Retargeting

ACM Classification Keywords
H.5.2. User Interfaces: Screen Design

HOW BIG IS YOUR SCREEN?
Websites are being viewed on a increasing diversity of devices. Among U.S. adults, 46% own smartphones, 57% have laptops, 19% own an e-book reader, and 19% have a tablet computer[5]. Across different manufacturers and models, these devices saturate a continuum of screen size, aspect ratio, and resolution. However, most websites are designed with fixed size layouts, (in the vicinity of 960px) frustrating visitors on mobile devices.

To cope with the challenge of adapting to a wide range of devices, designers are adopting a suite of techniques and strategies known as "responsive design."[4] Responsively designed webpages maintain a single document, whose layout "responds" to the viewport size and resolution by making discrete changes in the layout of page elements. For instance, in a responsive design a grid of photos is handled by progressively decreasing the number of columns in response to narrower display widths. (By contrast, proportional scaling results in shrinking the individual photos, limiting the effective size adaptation to a narrower range of sizes)

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s). Copyright is held by the author/owner(s).
UIST'14 Adjunct, October 5–8, 2014, Honolulu, HI, USA.
ACM 978-1-4503-3068-8/14/10.
http://dx.doi.org/10.1145/2658779.2658805

RETARGETING AS COMPILATION
ReMorph is a system for programmatically rewriting existing webpages, which we use to retrofit existing webpages with viewport-adaptive layouts. (i.e. responsive design)

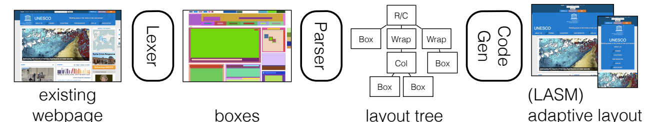

Figure 1. The ReMorph system design is loosely based on a compiler with lexer, parser and code generation stages.

Abandoning HTML/CSS
Lexing is responsible for extracting boxes from a webpage. Unlike most other systems that solve a similar problem, our lexer is designed for both analysis and synthesis; it ensures that we can independently reposition and resize the extracted boxes. First the lexer *embalms* the page to remove dynamic Javascript behavior, preserve the CSS-Rule-to-DOM-Node mapping (by rewriting the CSS selectors), and create CSS-style-closures. After embalming, the lexer determines which DOM Nodes have any visible effect on the page, (requires looking at over 10 independent CSS attributes) and which order boxes are drawn in, (by reimplementing the algorithm in Appendix E of the CSS 2.1 specification[2]) and finally reroots all visible DOM nodes as immediate children of the BODY element, sequenced in the correct draw order.

Parsing Visual Design Using NLP Algorithms
Parsing is responsible for hierarchically organizing the boxes output by the lexer. It produces a layout tree annotated with parameter values. Parsing must address (i) ambiguity about how boxes on the page should be grouped (Figure 2) and (ii) ambiguity about how those groups should respond when given more or less space in the layout. We use a 2d adaptation of the well known CYK parsing algorithm[3] from Natural Language Processing to resolve these ambiguities.

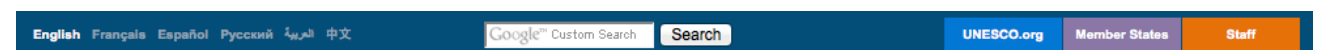

Figure 2. This horizontal sequence of boxes should be grouped into three subgroups of 6, 2, and 3 elements respectively, rather than one group of 11 elements.

CYK (aka. Chart) Parsing uses dynamic programming to solve subproblems in bottom-up order. Each sub-rectangle of the logical grid (Figure 3) defines a parsing sub-problem. Within this algorithmic structure, a grammar defines the structure of valid parses, while a scoring function (equivalently probability distribution) specifies which of the valid parses is the best choice. To solve a sub-problem, a problem's box is decomposed along a grid line (horizontally or

vertically) into two sub-problems; all applicable grammar rules are matched, and we keep the highest scoring parse for each type of symbol. (Our current vocabulary has three non-terminals: columns, rowcols, and wrappers) By leveraging features computed from the original webpage, the scoring function can bias the parse towards more desirable results. (Figure 4)

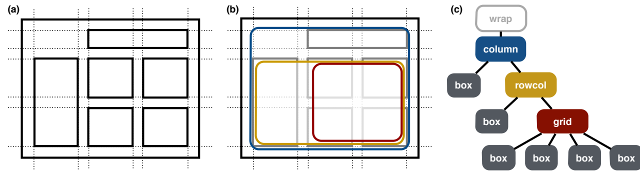

Figure 3. An example parsing problem, showing (a) the logical grid decomposition and one possible parse (b) in context and (c) abstractly.

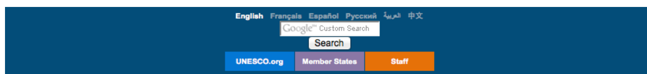

Figure 4. Using a simple feature (whether boxes come from the same HTML list) we can make desirable groupings more likely. As a result, the enclosing rowcol has switched to a column of 4 rows rather than a column of 11 items.

Generating Layout Assembly Code

Codegen is responsible for converting the layout tree from the parser into a layout program. Given a viewport width, this layout program sets the positions and sizes of all of the boxes on the page. We devised a small, easily implemented Layout Assembly (LASM) language in which to specify these programs. Each symbol in the layout tree is defined to expand into a particular chunk of LASM code. (Figure 6)

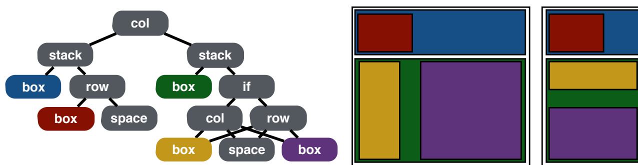

Figure 5. An example of a LASM program and the results of two executions with varying page widths

LASM programs execute (Figure 5) by recursively *placing* nodes by specifying a given input width; the resulting height is returned as output, along with the locations and sizes of all boxes in the sub-program. Row, Column, and Stack nodes sequence their child boxes horizontally, vertically, and in depth-order respectively. Box nodes cause a specific box to be positioned and sized, while Space nodes correspond to a visual no-op. Finally, If nodes choose which of their sub-trees to *place* depending on whether their input width is greater or less than a specified breakpoint.

LASM is deceptively simple. Formally, it is capable of encoding arbitrary piecewise-linear functions. This means it is capable of expressing layouts specified in most layout systems proposed to date, including sophisticated linear constraint systems such as the one proposed by Badros et al.[1]

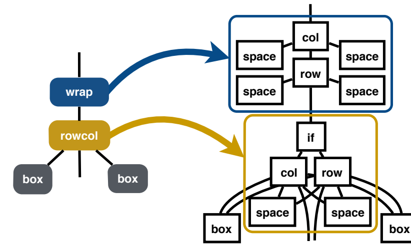

Figure 6. Code generation converts a layout tree into a LASM DAG by expanding nodes into chunks of LASM glued together using the same overall topology.

DESIGN TOOL OR AUTOMATION?

ReMorph can compute retargetings automatically, but we can also incorporate it into a design tool. To help explore this option, we conducted a preliminary study. We asked participants to retarget an existing website to a narrower width appropriate for mobile devices. While retargeting, our participants focused on the vertical order of page elements, while omitting elements they found less important. However our participants' decisions about the correct sequence or presence of page elements were mutually inconsistent. As a result, we are working on a design tool that (i) leverages ReMorph for a good starting proposal, (ii) makes re-sequencing & suppression of content easy, and (iii) relies on ReMorph to solve peripheral issues, like the setting of thousands of gutter, margin and other whitespace variables.

Currently the ReMorph system works on a limited set of test pages used for development. Moving forward, we plan to evaluate (a) how frequently ReMorph produces acceptable designs automatically, (b) how much effort is required of designers to edit the designs to their satisfaction, and (c) what kinds of designs are not addressed by ReMorph.

ACKNOWLEDGMENTS

Thanks to Pat Hanrahan and Ranjitha Kumar for their advice.

REFERENCES

1. Badros, G. J., Borning, A., Marriott, K., and Stuckey, P. Constraint cascading style sheets for the web. In *Proceedings of the 12th Annual ACM Symposium on User Interface Software and Technology*, UIST '99, ACM (New York, NY, USA, 1999), 73–82.
2. Bos, B., Çelik, T., Hickson, I., and Lie, H. W. Cascading style sheets level 2 revision 1 (css 2.1) specification. World Wide Web Consortium, Candidate Recommendation CR-CSS21-20070719, July 2007.
3. Jurafsky, D., and Martin, J. H. *Speech and Language Processing (2nd Edition) (Prentice Hall Series in Artificial Intelligence)*, 2 ed. Prentice Hall, 2008.
4. Marcotte, E. Responsive Web Design, May 2010. `http://alistapart.com/article/responsive-web-design`.
5. Zickuhr, K., and Smith, A. Digital Differences, April 2012. `http://pewinternet.org/Reports/2012/Digital-differences.aspx`.