Program Analyses for Interactive Web Services

Anders Møller
Claus Brabrand
Anders Sandholm
Michael I. Schwartzbach

Aarhus University
Overview

• The `<bigwig>` Project
  - motivation
  - main aspects
  - dynamic documents
  - data-flow analysis

• Gap&Field Analysis [POPL’00]
  - motivation
  - analysis specification

• Valid-HTML Analysis [PASTE’01]
  - summary graph construction and analysis
  - analysis specification
  - future work

• Conclusion
Web service construction today:

- **CGI scripts** (Perl/ C)
  - requires knowledge of technical details
  - mixing HTML and code becomes chaotic

- **ASP/ PHP/ JSP** (embedded scripts, interpreted by server)
  - nice for simple services
  - complex services are hard to maintain

- **Specialized servers** (e.g. Apache modules)
  - full control, efficient
  - low-level, expensive

General problems: no inherent session concept, no static guarantees!
The <bigwig> programming language

<bigwig>
high-level Web service specification

- HTML
- CGI / Apache
- JavaScript
- SSL
The `<bigwig>` programming language

`<bigwig>` = a type-safe Java/C-like core language
+ an inherent notion of `sessions`
+ **HTML** fragments as first-class values
+ a **relational database**
+ **concurrency control**
+ declarative **form-field validation**
+ information-flow and cryptographic **security**
+ **syntax-level** macros
Session-based services

```c
session Hello_1()
{
    int d;
    show <html>Hello World!</html>;
    d = getWeekDay();
    if (d==FRIDAY)
        show <html>You know it’s Friday?</html>;
    else
        show <html>It’s <i>not</i> Friday</html>;
}
```

- `show` sends an HTML document to client and waits for reply
- sessions provide **explicit control-flow!** (not like CGI/ PHP/ ...)

Program Analyses for Interactive Web Services
HTML as a Data Type

```plaintext
session Hello_2()
{
    string s;
    html H = <html>
        <h1>Hello World!</h1>
        Your name: <input type="text" name="f">
    </html>;
    show H receive [ s = f ];
    ...
}
```

- client input using `input` fields and `show-receive`
- HTML templates are values!
Constructing HTML documents

```plaintext
session Hello_3()
{
    html Cover = <html>
        <h1>Hello</h1>
        <![contents]> 
    </html>;
    html Greeting = <html><b>World</b></html>;
    html H = Cover <![contents = Greeting ];
    show H;
}
```

- \( x <[ g = y ] \) plugs \( y \) into \( x \) at the \( g \) gap
- a highly flexible method for building HTML documents!
The challenges

Given a `<bigwig>` program, check statically whether

• at every `x [<bigwig> g = y ]`
  - `x` always has a gap named `g` ?
  - are `x` and `y` always “compatible” w.r.t. gaps and fields ?

• at every `show x receive [ z = f, ... ]`
  - `x` always is a well-formed, valid HTML document
    (according to W3C’s specification) ?
  - every input field `f` being received always occurs in `x` ?
  - the type of `z` always matches that of `f` ?

Neither Perl, C, Java, PHP, ASP, or JSP can achieve that!
- but `<bigwig>` makes it all possible!
The solution

Apply standard, **data-flow analysis** techniques, but with highly **domain-specific lattices**
A data-flow analysis framework

Given

• the **control-flow graph** of the program,

• a finite **lattice** describing the abstract values, and

• monotone **transfer functions** for the various statements,

find the least solution using fixed-point iteration.

The result describes a **conservative approximation** of what values can occur where.
The dyndoc subset of \textlangle\textit{bigwig}\textrangle

Statements:

- \textit{x}_1 = \textit{x}_2; \quad \text{template variable assignment}
- \textit{x} = \textlangle html\rangle c \textrangle \textit{/html}\rangle \quad \text{template constant assignment}
- \textit{y} = “s”; \quad \text{string variable assignment}
- \textit{x}_1 = \textit{x}_2 \langle [ g = \textit{x}_3 \rangle \quad \text{template gap plugging}
- \textit{x}_1 = \textit{x}_2 \langle [ h = \textit{y} \rangle \quad \text{string gap plugging}
- \text{show} \times \text{receive} [ z = f, \ldots ] \quad \text{client interaction}

HTML fragments:

\[
\begin{align*}
\textit{c} & ::= \langle [ \textit{g} \rangle  \\
& | \langle \text{tag a}\ast\rangle \textit{c} \langle /\text{tag}\rangle  \\
& | \textit{c} \textit{c}  \\
\textit{a} & ::= \textit{n} = “s”  \\
& | \textit{n} = [ \textit{h} ]
\end{align*}
\]

- template gap
- element with attribute and contents
- sequence
- normal attribute
- attribute gap

Special HTML elements: \texttt{input}, \texttt{select}, ...
A control-flow graph: Hello_1

x = <html>Hello World!</html>

show x

skip /* d = getWeekDay() */

skip /* if (d==FRIDAY) */

x = <html>You know it’s Friday?</html>

x = <html>It’s <i>not</i> Friday</html>

show x

show x

join
Lattices for Gap & Field analysis [POPL’00]

\[
\text{Env} = \text{HtmlVar} \rightarrow (\text{GapName} \rightarrow \text{GapKind}) \times (\text{FieldName} \rightarrow \text{FieldKind})
\]

Transfer function for statement S: \( T_S : \text{Env} \rightarrow \text{Env} \)
Valid-HTML analysis [PASTE’01]

Two main parts:

1. a data-flow analysis providing a **summary graph** for every template variable and program point

2. a validation of each summary graph with respect to an **abstract description of XHTML**

(XHTML is easier to work with than HTML, but is basically the same.)
Summary Graphs

Summary graph: a graph representing all the ways templates can be plugged together

- **Node**  ~  template constant
- **Edge**  ~  template plug
- **Node label**  ~  attribute plug
- **Root node**  ~  outermost template

```html
<h1>Hello</h1>
```

```html
<a href='http://www.brics.dk/'>World</a>
```

url = http://www.brics.dk/
session Hello_3()
{
    html Cover = <html>
        <h1>Hello</h1>
        <[contents]>
    </html>;
    html Greeting = <html><b>World</b></html>;
    html H = Cover <[ contents = Greeting ];
    show H;
}
Summary Graph inference

• The set of summary graphs forms a lattice \((SG, \subseteq)\)

• (Quite complex) transfer functions can be defined

• A data-flow analysis returns a result of the form:

\[
\text{ProgramPoint} \rightarrow \text{HtmlVar} \rightarrow \text{SG}
\]
Transfer function for template plug

\[ x_1 = x_2 < \begin{array}{|c|} g = x_3 \end{array} \]

is modeled by assigning a new summary graph to \( x_1 \) consisting of:

- the union of the **template nodes** from \( x_2 \) and \( x_3 \)
- the **roots** from \( x_2 \)
- the union of the **attribute labels** of \( x_2 \) and \( x_3 \)
- the union of the **plug edges** in \( x_2 \) and \( x_3 \) plus edges from each node in \( x_2 \) that may have an open \( g \) gap to each root in \( x_3 \)
Gap Track analysis

- but how do we know which nodes in $x_2$ that may have open $g$ gaps?

Solution: yet another data-flow analysis

A simple **Gap Track analysis** tracks the origins of gaps:

ProgramPoint $\rightarrow$ HtmlVar $\rightarrow$ GapName $\cup$ FieldName $\rightarrow$ $2^{\text{Template}}$
Validating summary graphs

Goal: check that all documents that can be generated from a given summary graph are valid XHTML 1.0 (≈ HTML 4.01)

The algorithm:
• traverse the summary graph, compare each node with the requirements in an “abstract DTD”
• apply memoization

- this phase is both sound and complete!
- bonus: a simple but powerful XML schema language
Abstract DTDs — a simple schema language

An **Abstract DTD** consists of **element** declarations, each containing:
- an element **name**
- **attribute** and **content** declarations
- a **constraint**: a boolean expression of
  - $\text{attr}(a)$ - “require attribute presence”
  - $\text{content}(c)$ - “require content presence”
  - $\text{order}(\{c_1,\ldots,c_n\}, \{c_1',\ldots,c_m'\})$ - “require content ordering”
  - $\text{value}(a, \text{regexp})$ - “require attribute value”

defining a **set of valid XML trees** in a top-down manner.

(Examples: XHTML, WML, VoiceXML, ...)
Experiments: Valid-HTML analysis

<table>
<thead>
<tr>
<th>Program</th>
<th>Lines</th>
<th>Fragments</th>
<th>Time (sec.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>chat</td>
<td>65</td>
<td>3</td>
<td>0.1</td>
</tr>
<tr>
<td>guess</td>
<td>75</td>
<td>6</td>
<td>0.1</td>
</tr>
<tr>
<td>calendar</td>
<td>77</td>
<td>5</td>
<td>0.1</td>
</tr>
<tr>
<td>xbiff</td>
<td>561</td>
<td>18</td>
<td>0.1</td>
</tr>
<tr>
<td>webboard</td>
<td>1132</td>
<td>37</td>
<td>0.6</td>
</tr>
<tr>
<td>cdshop</td>
<td>1709</td>
<td>36</td>
<td>0.5</td>
</tr>
<tr>
<td>jao0</td>
<td>1941</td>
<td>73</td>
<td>2.4</td>
</tr>
<tr>
<td>bachelor</td>
<td>2535</td>
<td>137</td>
<td>8.2</td>
</tr>
<tr>
<td>courses</td>
<td>4465</td>
<td>57</td>
<td>1.3</td>
</tr>
<tr>
<td>eatcs</td>
<td>5345</td>
<td>133</td>
<td>6.7</td>
</tr>
</tbody>
</table>

( after macro expansion )

( Linux / 800MHz Pentium III )

- many validation errors found, no spurious errors!
--- brics.wig:24: HTML validation:

brics.wig:4:
  warning: illegal attribute ‘bgcolo’ in ‘body’
  template: <body bgcolo=[color]><form>...</form></body>

brics.wig:5:
  warning: possible illegal subelement ‘td’ of ‘table’
  template: <table><[contents]></table>
  contents: td
  plugs: contents:{brics.wig:22}

brics.wig:10:
  warning: possible element constraint violation at ‘br’
  template: <br clear=[clear]/>
  constraint: value(clear,{left,all,right,clear,none})
  plugs: clear:{brics.wig:23}
“Why not just use standard PL techniques?”

“An XML document is just a **tree**, so simply

- use **recursive types** instead of **abstract DTDs**, and
- use **constructor functions** instead of the **gap/plug** approach”
“Why not just use standard PL techniques?”

Bad idea!

• **abstract DTDs** are strictly more expressive than **recursive types**
  - example: the `<title>` element in `<head>` in HTML

• the **gap/plug** approach is strictly more expressive than **constructor functions**
  - we can do “non-local” plugs instead of building the trees bottom-up only

— and we need this extra power in practice!
Summary

• Increasing need for high-level programming language for making interactive Web services

• Two central ideas in <bigwig>:
  - the session model, show/ receive
  - flexible but safe, dynamic construction of Web pages:
    • gap/plug document construction
    • gap&field safety
    • always showing valid HTML
Summary

- Gap analysis
- Field analysis
- HTML analysis
  - Gap Track analysis
  - Summary Graph inference
  - Summary Graph validation wrt. XHTML DTD

Data-flow analyses

Summary graph analysis
Papers from the `<bigwig>` project

- A Runtime System for Interactive Web Services  [WWW8]
- Distributed Safety Controllers for Web Services  [FASE’98]
- PowerForms: Declarative Client-Side Form Field Validation  [WWW Journal]
- Growing Languages with Metamorphic Syntax Macros  [submitted]
- Language-Based Caching of Dynamically Generated HTML  [submitted]
- The `<bigwig>` Project  [submitted]
- A Type System for Dynamic Documents  [POPL’00]
- Static Validation of Dynamically Generated HTML  [PASTE’01]
Conclusion

Future work:
- full XML support (with schemas as types)
- support for Web site construction
- JavaWig: integrating $\texttt{<bigwig>}$ into Java
- other aspects of Web programming...

More information:

http://www.brics.dk/bigwig/

- Open Source implementation
- documentation and examples
- research papers