A Type System for Dynamic Web Documents

Anders Sandholm and Michael I. Schwartzbach

BRICS, Department of Computer Science
University of Aarhus, Denmark
{sandholm,mis}@brics.dk

Abstract

Many interactive Web services use the CGI interface for communication with clients. They will dynamically create HTML documents that are presented to the client who then resumes the interaction by submitting data through incorporated form fields. This protocol is difficult to statically type-check if the dynamic documents are created by arbitrary script code using `printf`-like statements. Previous proposals have suggested using static document templates which trades flexibility for safety. We propose a notion of typed, higher-order templates that simultaneously achieve flexibility and safety. Our type system is based on a flow analysis of which we prove soundness. We present an efficient runtime implementation that respects the semantics of only well-typed programs. This work is fully implemented as part of the `<bigwig>` system for defining interactive Web services.

1 Introduction

The HTTP protocol was originally designed for browsing *static* documents connected with hyperlinks. CGI together with HTML forms allows *dynamic* creation of documents whose contents are constructed on the server at the time the document is requested [5].

Interactive Web services make intensive use of dynamic documents when communicating with Web clients. A standard way of implementing such services is through CGI-scripts (often written in Perl [11]) that dynamically generate documents simply by printing their contents directly to standard output.

This technique is fraught with dangers, since it is impossible to statically verify that an arbitrary script will output a legal and consistent HTML document. First, the presented document may be garbled if the generated output does not contain properly balanced HTML tags. Second, each presented document is typically an HTML form that prompts the client for input and contains a submit button that allows the service to resume execution. If the input fields are dynamically generated, then they might not correspond to those that the resuming service expects.

There are alternatives to the CGI protocol, such as ASP [4], PHP [2], and Java Servlets [6], but the protocol remains popular since it is the only mechanism that is ubiquitously supported by all servers on all platforms. Thus, it remains relevant to provide static safety for CGI-programs.

The domain-specific language MAWL provides high-level mechanisms for programming interactive Web services [1, 8]. The problems with dynamic documents are addressed through the introduction of static document templates that behave quite similarly to typed functions. The arguments are integer or string values that are plugged into named gaps in the template before it is presented. The result from the client is a tuple of simple values corresponding to the input fields that the client is requested to fill in. The simple type system of MAWL then guarantees that the templates are used correctly.

On the downside, MAWL has sacrificed much flexibility in order to obtain safety. For many uses the document templates are too static, since they are essentially constant documents apart from the fixed collection of gaps that can be plugged with simple values. MAWL partially alleviates this shortcoming by allowing special iteration gaps that can be filled with lists of values. However, MAWL cannot construct a dynamic document that uses nested lists to display an unbounded hierarchical structure such as the active threads in a bulletin board or the directory of a file system.

The language Guide [9], which is inspired by MAWL, uses a similar template mechanism with gaps that are plugged with simple values, but without static type checking.

In this paper we propose a notion of *higher-order* document templates that generalize those of MAWL and Guide while preserving static safety.

Documents are first-class values that may be computed and stored in variables. A document may contain named gaps that are placeholders for either HTML fragments or string attributes in tags. Such gaps may at runtime be plugged with concrete values that are themselves templates. Since those values may contain further gaps, this is a highly dynamic mechanism for building documents reminiscent of higher-order functions [7]. As input fields may now appear dynamically, they are much harder to keep track of.

To allow a sufficiently flexible use, we introduce a sophisticated notion of document types and employ a flow-sensitive type checker. As a further benefit, we introduce an efficient runtime representation that is sound for only well-typed programs, for which it represents document values with almost perfect sharing in such a way that the plug operation takes constant time only while the show command remains efficient.

Our proposal has been fully implemented in the `<bigwig>` project [3] which like MAWL provides a high-level language for specifying interactive Web services. This paper also provides a case study in using the flow analysis framework of [10] in a domain-specific setting.

In Section 2 we give a stylized version of a language with

Permission to make digital or hard copies of all or part 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. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
POPL 2000 Boston MA USA
Copyright ACM 2000 1-58113-125-9/00/1...$5.00

dynamic documents; Section 3 describes the notion of document types; Section 4 presents and proves correctness of the flow-based type checker and proves its correctness; and Section 5 specifies the efficient runtime implementation.

2 The Language

We consider a simple imperative language DynDoc which corresponds to the minimal fragment of <bigwig> that deals with dynamic documents. In examples, we will allow ourselves to use a richer syntax drawn from the full language, which uses a C-like syntax.

2.1 Syntax

We first give a description of the extended HTML syntax that we use. From that we define the central notion of a well-formed document. Then, the abstract syntax of our simple imperative language DynDoc is given.

Gaps The HTML syntax that we use is extended in two ways. First, we introduce two kinds of named *gaps*:

```
This <[foo]> <b>is <[bar]></b> you know!
Some pictures <img src=[baz]> of funny ducks.
```

Here, foo and bar are names of gaps that can be plugged with HTML fragments, while baz is a gap that can be plugged with an attribute string value.

An HTML document constant, *const*, is thus ideally a tree structure conforming to the following abstract syntax:

$$const ::= \text{<}tag\ attr^*\text{>}\ const^*\ \text{</}tag\text{>} \mid \text{<}[y]\text{>} \mid s$$
$$attr ::= name\text{=}value \mid name\text{=}[y]$$

where *tag* ranges over HTML tags (the opening and closing tags must match), *name* over attribute names, *value* over attribute values, and y over gap names. An HTML document is thus a tree structure where the nodes consist of a tag name, an attribute list of named values or gaps, and a number of sub-trees. The leaves are either strings or named gaps. Note that in the examples, the concrete syntax is not quite as restrictive, such as the example above, where the img tag does not have a closing tag.

Tuples Second, we extend the HTML form input fields with a mechanism for collating input values:

```
<tuple name="order">
  Pizza Napoli
  <input type="hidden" name="pizza" value="napoli">
  <br>
  Amount <input type="text" name="amount"><br>
  Extra cheese?
  Yes <input type="radio" name="cheese" value="yes">
  No <input type="radio" name="cheese" value="no">
</tuple>
<p>
<tuple name="order">
  Pizza Roma
  <input type="hidden" name="pizza" value="roma">
  <br>
  Amount <input type="text" name="amount"><br>
  Extra cheese?
  Yes <input type="radio" name="cheese" value="yes">
  No <input type="radio" name="cheese" value="no">
</tuple>
```

The tuples are viewed as a single input field named order. This extension allows multiple occurrences of the same input field name to appear in a document. When the field values are received, they are automatically transformed into tuples in a relation with schema {pizza, amount, cheese}. If the tuple occurs only once in a document, the field values are transformed into a single tuple. Tuples cannot be nested and are not allowed to contain checkboxes or HTML gaps.

These two HTML extensions are purely internal to the DynDoc language (and to <bigwig>); only standard documents are presented to the Web client.

Well-formed HTML Throughout this paper we will be working with the notion of well-formed documents. A document, *const*, is *well-formed* if the following requirements are satisfied. All gap names occur at most once in *const*. If an input field name occurs more than once, all the occurrences must be of the same kind, which has to be either radio, checkbox, or tuple. If of kind tuple, the occurrences must have the same associated schema. In the following, **HTML** denotes the set of well-formed HTML documents.

Abstract syntax For the formal presentation we introduce the following syntactic categories:

$b \in$ **BExp**	boolean expressions
$d \in$ **DExp**	document expressions
$s \in$ **StrExp**	string expressions
$S \in$ **Stm**	statements

Boolean and string expressions will not be further defined whereas **DExp** and **Stm** will be defined later by a grammar.

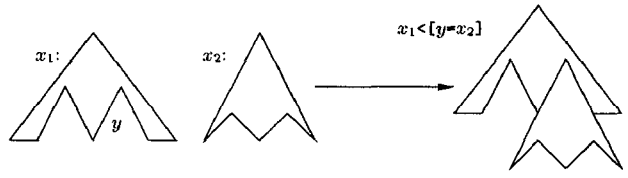
For a given program S_* the following finite sets of program variables, document variables, gap names, and field names are given:

$v \in$ **PVar**	program variables
$y \in$ **GName**	gap names
$x \in$ **DVar**	document variables
$z \in$ **FName**	field names

The syntax of DynDoc is completed by the following *abstract syntax* defining document expressions and statements:

$$\begin{aligned} d ::= &\ x \mid const \mid x_1\text{<}[y = x_2] \mid x\text{<}[y = s] \\ S ::= &\ [x = d]^l \mid [\texttt{skip}]^l \mid S_1\,;\,S_2 \mid \texttt{if}\,[b]^l\,S_1\,\texttt{else}\,S_2 \mid \texttt{while}\,[b]^l\,S \\ \mid &\ [\texttt{show}\,x\,\texttt{receive}\,[v_1 = z_1, \ldots, v_n = z_n]]^l \end{aligned}$$

A document expression can be a document variable, a constant HTML document with zero or more named gaps, or a plug operation. The result of the plug operation $x_1\text{<}[y = x_2]$ is the value of x_1 where its y gap is replaced by the value of x_2:



There is no problem in having the plug operation accept document expressions rather than document variables. However, to simplify the presentation we have chosen to work with simpler, normalized document expressions.

A statement is either a document variable assignment, a `skip` statement, a sequence of statements, a conditional, a `while` loop, or a `show` call. The `show` call $[\texttt{show}\, x\, \texttt{receive}\, [v = z]]^l$ allows interaction with the client: The contents of the document variable x is shown to the client, and when the client submits the reply, the received value of the form field named z is assigned to the program variable v. Again, we require for simplicity only that the argument to `show` is a document variable rather than a document expression.

Later, we will associate document type flow information with assignment statements, tests (in conditionals and loops), `skip`, and `show` calls. For simplicity, we will assume that these *elementary blocks* are initially assigned distinct labels, $l \in \mathbf{Lab}$. The occurrence of $[\cdot]^l$ in the syntax above denotes this explicit labeling.

Note that this is not a complete language, but merely a sketch suitable for formal analysis. The `skip` statement can be thought of as representing other statements that are not relevant for type-checking documents.

Examples An example of the plug operation in `<bigwig>` syntax is:

```
{ const html H = <html bgcolor=[color]>
                    Hello <[what]>!
                 </html>;
  show H<[color="red",what="world"];
}
```

Here, `color` is a string gap and `what` is an HTML gap. Another standard example builds an HTML table from a vector of tuples:

```
{ schema Student {string name; int age};
  vector Student S;
  const html Row = <html><tr><td><[name]></td>
                              <td><[age]></td>
                          </tr>
                          <[row]>
                      </html>;
  html T = <html><table><[row]></table></html>;
  int i;
  for (i=0; i<|S|; i++)
    T = T<[row = Row<[name=S[i].name,age=S[i].age]];
  show T;
}
```

2.2 Semantics

For the purpose of giving a formal small-step operational semantics of DynDoc we shall first define a number of categories and functions.

First define a *state* as a mapping from document variables to well-formed HTML documents:

$$\sigma \in \mathbf{State} = \mathbf{DVar} \to \mathbf{HTML}.$$

Note that for simplicity, program variables are not considered part of the state. To deal with boolean, string, and document expressions we require the following three semantic functions

$$\begin{aligned} \mathrm{B} &: \mathbf{BExp} \to \mathbf{State} \to \{\mathsf{true}, \mathsf{false}\}, \\ \mathrm{S} &: \mathbf{StrExp} \to \mathbf{State} \to \mathbf{String}, \text{ and} \\ \mathrm{D} &: \mathbf{DExp} \to \mathbf{State} \hookrightarrow \mathbf{HTML}, \end{aligned}$$

where **String** denotes the set of strings. We assume B and S are given and define the partial map D below.

Semantics of document expressions To give semantics to document expressions we first need to define functions that will allow us to inspect the gaps and input fields of a document. For this we define the set $\mathbf{GKind} = \{\mathsf{nogap}, \mathsf{html}, \mathsf{string}, \mathsf{error}\}$ to be the set of gap kinds. From that we can define the function

$$gap : \mathbf{HTML} \to \mathbf{GName} \to \mathbf{GKind}$$

which maps well-formed HTML documents into a map that associates gap names with gap kinds. The *gap* function is defined inductively as follows:

$$\begin{aligned} &gap(\texttt{<}tag\ name\texttt{=}[y]\texttt{>}\ const_1 \ldots const_n\ \texttt{</}tag\texttt{>}) = \\ &\qquad [y \mapsto \mathsf{string}] \oplus_g gap(const_1) \oplus_g \cdots \oplus_g gap(const_n) \\ &gap(\texttt{<}tag \ldots \texttt{>}\ const_1 \ldots const_n\ \texttt{</}tag\texttt{>}) = \\ &\qquad [\,] \oplus_g gap(const_1) \oplus_g \cdots \oplus_g gap(const_n) \\ &gap(\texttt{<}[y]\texttt{>}) = [y \mapsto \mathsf{html}] \\ &gap(s) = [\,] \end{aligned}$$

where $[\,]$ is the function that maps everything to nogap, $[y \mapsto kind]$ the function that maps everything but y (which is mapped to *kind*) to nogap, where *kind* is any gap kind, and $\oplus_g$ on gap kinds is defined as follows (and is lifted pointwise to gap maps):

$$a \oplus_g b = \begin{cases} a & \text{if } b = \mathsf{nogap}, \\ b & \text{if } a = \mathsf{nogap}, \\ \mathsf{error} & \text{otherwise.} \end{cases}$$

Note that by definition of well-formedness, *gap* is a well-defined function, that is, the error case of $\oplus_g$ will never occur for *gap*. Thus for a given well-formed document d and a gap name y, $gap(d)(y)$ gives the unique kind (string or html) of the y gap in d and if d does not have a y gap, the result is nogap. Similarly we consider the set of input field kinds:

nofield	field name not associated with a field
ordinary	select menus, text, textarea, and hidden fields
radio	radio buttons
checkbox	check boxes and select multiple
tuple(F_i)	single tuple field with schema F_i
rel(F_i)	more than one tuple field with schema F_i

where $F_1, \ldots, F_n$ are the schemas occurring in S_*. Again, we shall use the error kind to indicate lack of well-formedness. Let **FKind** denote the set of field kinds. We can then define the function

$$field : \mathbf{HTML} \to \mathbf{FName} \to \mathbf{FKind}$$

such that *field*(d)(z) is the field kind of the input field named z in the document d. More precisely, *field* is defined inductively as follows:

$$\begin{aligned} &field(\texttt{<input name="}z\texttt{" type="radio"></input>}) \\ &\quad = [z \mapsto \mathsf{radio}] \\ &field(\texttt{<input name="}z\texttt{" type="checkbox"></input>}) \\ &\quad = [z \mapsto \mathsf{checkbox}] \\ &field(\texttt{<input name="}z\texttt{" type=...></input>}) \\ &\quad = [z \mapsto \mathsf{ordinary}] \end{aligned}$$

$$
\begin{aligned}
&\mathit{field}(\texttt{<select name="}z\texttt{">}\ \mathit{const}_1 \ldots \mathit{const}_n\ \texttt{</select>})\\
&\quad = [z \mapsto \mathsf{ordinary}] \oplus_f \mathit{field}(\mathit{const}_1) \oplus_f \cdots \oplus_f \mathit{field}(\mathit{const}_n)\\
&\mathit{field}(\texttt{<tuple name="}z\texttt{">}\ \mathit{const}_1 \ldots \mathit{const}_n\ \texttt{</tuple>})\\
&\quad = [z \mapsto \mathsf{tuple}((\mathit{field}(\mathit{const}_1) \oplus_f \cdots \oplus_f \mathit{field}(\mathit{const}_n))^{-1}\\
&\qquad\qquad (\{\mathsf{ordinary}, \mathsf{radio}\}))]\\
&\mathit{field}(\texttt{<}\mathit{tag}\texttt{>}\ \mathit{const}_1 \ldots \mathit{const}_n\ \texttt{</}\mathit{tag}\texttt{>})\\
&\quad = \mathit{field}(\mathit{const}_1) \oplus_f \cdots \oplus_f \mathit{field}(\mathit{const}_n)\\
&\mathit{field}(\texttt{<[}y\texttt{]>}) = \mathit{field}(s) = [\,]
\end{aligned}
$$

where the $\oplus_f$ operation on field kinds is defined as follows (again we can easily lift to field maps):

$$
a \oplus_f b = \begin{cases} \mathsf{rel}(F_i) & \text{if } a, b \in \{\mathsf{tuple}(F_i), \mathsf{rel}(F_i)\},\\ a & \text{if } a = b \in \{\mathsf{radio}, \mathsf{checkbox}\},\\ a & \text{if } b = \mathsf{nofield},\\ b & \text{if } a = \mathsf{nofield},\\ \mathsf{error} & \text{otherwise.} \end{cases}
$$

Again, by definition of well-formedness $\mathit{field}(d)(z)$ will never be error for a well-formed document d. Notice that if there are several occurrences of a tuple z (with schema F), the kind supplied by the *field* function for z is $\mathsf{rel}(F)$. Otherwise, that is, if there is only one occurrence, the kind is $\mathsf{tuple}(F)$. The remaining cases (nofield, ordinary, radio, and checkbox) are straightforward.

With the *gap* and *field* functions in place, we can now define the semantic function D:

$$
\begin{aligned}
\mathrm{D}[\![x]\!]\sigma &= \sigma(x)\\
\mathrm{D}[\![\mathit{const}]\!]\sigma &= \mathit{const}\\
\mathrm{D}[\![x\texttt{<[}y = x_2\texttt{]}]\!]\sigma &= \mathit{plug}(\sigma(x_1), y, \sigma(x_2))\\
\mathrm{D}[\![x\texttt{<[}y = s\texttt{]}]\!]\sigma &= \mathit{strplug}(\sigma(x_1), y, \mathrm{S}[\![s]\!]\sigma),
\end{aligned}
$$

where $\mathit{plug}(d_1, y, d_2)$ is defined if and only if

- $\mathit{gap}(d_1)(y) = \mathsf{html}$;
- $\forall y' : (\mathit{gap}(d_1)[y \mapsto \mathsf{nogap}] \oplus \mathit{gap}(d_2))(y') \neq \mathsf{error}$; and
- $\forall z : (\mathit{field}(d_1) \oplus \mathit{field}(d_2))(z) \neq \mathsf{error}$.

If defined $\mathit{plug}(d_1, y, d_2)$ is the well-formed document resulting from replacing the y gap in d_1 with d_2. Intuitively, the first requirement ensures that the gap to be replaced is in fact there and the remaining two requirements ensure that the resulting document will be well-formed. Similarly, $\mathit{strplug}(d, y, s)$ is defined if and only if $\mathit{gap}(d)(y) \in \{\mathsf{html}, \mathsf{string}\}$ in which case $\mathit{strplug}(d, y, s)$ is just d_1 with its y gap replaced by s.

Thus the meaning of a document variable is its associated value in the current state. The meaning of a constant is just the well-formed document constant itself. The meaning of $x_1\texttt{<[}y = x_2\texttt{]}$ is defined at σ only if $\sigma(x_1)$ has an HTML gap named y and if the resulting document is well-formed, that is, if the maps combining the gap and field maps of the two operands are total maps, that is, if they do not map any values to error. Similarly, $x\texttt{<[}y = s\texttt{]}$ is defined on σ if and only if $\sigma(x_1)$ has a y gap.

Semantics of statements Before we can define the configurations of our semantics and the semantic rules for statements we need to define the map:

$$\mathit{typeof} : \mathbf{PVar} \to \mathbf{PType}$$

which assigns a type to each of the program variables, that is, to each of the non-document variables of the program. The set of program variable types is the following: $\mathbf{PType} = \{\mathsf{string}, \mathsf{tuple}(F_1), \mathsf{rel}(F_1), \ldots, \mathsf{tuple}(F_n), \mathsf{rel}(F_n)\}$. Such types are native to the <bigwig> language.

We choose to use a small step semantics. This will allow us to reason about intermediate stages in a program execution (even for non-terminating programs). A *configuration* of the semantics is either a pair consisting of a statement and a state or it is simply a state in which case the configuration is called terminal. The *transitions* of the semantics, expressing how the configuration is changed by one step of computation, are of the form $\langle S, \sigma\rangle \to \langle S', \sigma'\rangle$ and $\langle S, \sigma\rangle \to \sigma$. Non-terminal configurations with no enabled transitions are called *stuck* configurations. In the semantics presented below, stuck configurations are reached only when errors occur in the execution. We would thus like to avoid stuck configurations, that is, for the type system that we will introduce later, we expect the property that type-checked programs will never enter stuck configurations.

The detailed definition of the semantics of statements is defined in Figure 1. We write $\sigma[x \mapsto d]$ for the state that is as σ except

$$
\begin{aligned}
&[\mathit{asn}] && \langle [x = d]^l, \sigma\rangle \to \sigma[x \mapsto \mathrm{D}[\![d]\!]\sigma]\\
& && \quad \text{if } \mathrm{D}[\![d]\!]\sigma \text{ is defined}\\
&[\mathit{skip}] && \langle [\texttt{skip}]^l, \sigma\rangle \to \sigma\\
&[\mathit{seq}_1] && \frac{\langle S_1, \sigma\rangle \to \langle S_1', \sigma'\rangle}{\langle S_1\,;\,S_2, \sigma\rangle \to \langle S_1'\,;\,S_2, \sigma'\rangle}\\
&[\mathit{seq}_2] && \frac{\langle S_1, \sigma\rangle \to \sigma'}{\langle S_1\,;\,S_2, \sigma\rangle \to \langle S_2, \sigma'\rangle}\\
&[\mathit{if}_1] && \langle \texttt{if}\,[b]^l\,S_1\,\texttt{else}\,S_2, \sigma\rangle \to \langle S_1, \sigma\rangle\\
& && \quad \text{if } \mathrm{B}[\![b]\!]\sigma = \mathsf{true}\\
&[\mathit{if}_2] && \langle \texttt{if}\,[b]^l\,S_1\,\texttt{else}\,S_2, \sigma\rangle \to \langle S_2, \sigma\rangle\\
& && \quad \text{if} \mathrm{B}[\![b]\!]\sigma = \mathsf{false}\\
&[\mathit{wh}_1] && \langle \texttt{while}\,[b]^l\,S, \sigma\rangle \to \langle (S;\,\texttt{while}\,[b]^l\,S), \sigma\rangle\\
& && \quad \text{if } \mathrm{B}[\![b]\!]\sigma = \mathsf{true}\\
&[\mathit{wh}_2] && \langle \texttt{while}\,[b]^l\,S, \sigma\rangle \to \sigma\\
& && \quad \text{if } \mathrm{B}[\![b]\!]\sigma = \mathsf{false}\\
&[\mathit{show}] && \langle [\texttt{show}\,x\,\texttt{receive}\,[v_1 = z_1, \ldots, v_n = z_n]]^l, \sigma\rangle \to \sigma\\
& && \quad \text{if } \forall i : \mathit{typeof}(v_i) \lessdot \mathit{field}(\sigma(x))(z_i) \text{ and}\\
& && \quad \forall z \in \mathbf{FName} \setminus \{z_1, \ldots, z_n\} : \mathit{field}(\sigma(x))(z) = \mathsf{nofield}
\end{aligned}
$$

Figure 1: The semantics of statements.

that x is mapped to d and the relation $\lessdot \subseteq \mathbf{PType} \times \mathbf{FKind}$ is defined as follows: $\mathsf{string} \lessdot \mathsf{radio}$, $\mathsf{string} \lessdot \mathsf{ordinary}$, $\mathsf{tuple}(F) \lessdot \mathsf{tuple}(F)$, $\mathsf{rel}(F) \lessdot \mathsf{nofield}$, $\mathsf{rel}(F) \lessdot \mathsf{tuple}(F)$, $\mathsf{rel}(F) \lessdot \mathsf{rel}(F)$, and $\mathsf{rel}(\{\mathit{name}\}) \lessdot \mathsf{checkbox}$. Note also, that due to the fact that program variables are not considered part of the state, the state σ remains the same after the execution of a `show-receive` statement in [*show*].

That is, if the receiving variable is of type string, the corresponding field must be of kind radio or ordinary. If the variable is of type tuple, the field must be so as well, whereas if the variable is of type relation, then the field can either not appear in the doc-

ument or appear as a tuple or a relation. If the variable is of kind relation with a singleton schema, the corresponding field can be a checkbox.

Notice, that the stuck configurations are caused either by $\mathrm{D}[\![d]\!]\sigma$ not being defined in [*asn*], that is, by an illegal plug operation, or by non-matching program variables and document fields in [*show*].

Intuitively, we wish to devise a type checker that allows as liberal a use of dynamic documents as possible, while guaranteeing that no errors occur, that is, while guaranteeing the following requirements:

- in $x_1 < [y = x_2]$, x_1 must have an HTML gap named y;
- in $x < [y = s]$, x must have a gap named y;
- all constructed documents must be well-formed; and
- in a `show` call, the document fields must match the `receive` program variables.

3 Document Types

We introduce a somewhat complicated notion of document type. These types are not explicitly written by the programmer, but are inferred by the compiler using a global flow analysis.

A document type has two distinct components: a *gap map* and a *field map*. The gap map describes the gaps that are present in a document and, similarly, the field map describes its input fields. To facilitate the subsequent flow analysis, these maps are defined as points in two finite lattices.

3.1 Gap Maps

To each gap name in **GName** we associate for a given document value a gap kind. This is html if this name denotes an HTML gap, string if it denotes a string gap, and nogap if it is not associated with a gap. Furthermore, we extend **GKind** with the extra kinds $\perp$ (no information available yet) and error (an error has occurred). We impose a partial ordering on gap kinds thus making **GKind** a finite lattice (see Figure 2).

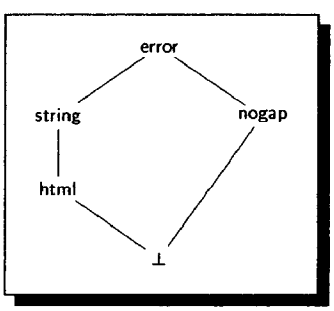


Figure 2: **GKind**, the finite lattice of gap kinds.

This particular ordering is carefully chosen to ensure that functions defined in Section 4 become monotone; intuitively, higher values describe more restricted capabilities of gaps. From **GKind** we define the notion of a *gap map*: $\mathbf{GMap} = \mathbf{GKind}^{|\mathbf{GName}|}$. Though elements of **GMap** are $|\mathbf{GName}|$-tuples, we shall sometimes use the more intuitive function notation. That is, a gap map can be considered a function from **GName** to **GKind**. Gap maps are associated with document values and variables.

3.2 Field Maps

To each field name in **FName** we associate for a given document value a field kind that describes its presence and capabilities. It should be clear that our approach is quite flexible; for example, we also capture the `tuple` extension that we have introduced in `<bigwig>`.

We impose a partial order on the field kinds described earlier extended with a bottom element, $\perp$. Again, the ordering is carefully chosen to ensure that functions defined in Section 4 become monotone. The field kinds thus form a finite lattice, **FKind** (see Figure 3). As for gaps, we lift to field maps, $\mathbf{FMap} = \mathbf{FKind}^{|\mathbf{FName}|}$.

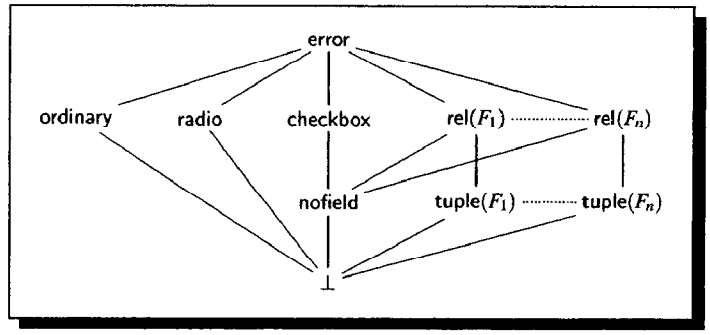


Figure 3: **FKind**, the finite lattice of field kinds.

3.3 Type environments

We shall define $\mathbf{DType} = \mathbf{GMap} \times \mathbf{FMap}$ to be the set of document types. A document type is thus a pair of a gap and a field map. Document types are associated with document values and variables. Given a document type, we need projection maps to extract the gap map and field map of the type: $\pi_g : \mathbf{DType} \rightarrow \mathbf{GMap}$ and $\pi_f : \mathbf{DType} \rightarrow \mathbf{FMap}$. Also, we shall denote by $gap \otimes field$ the map $(gap \otimes field)(d) = (gap(d), field(d))$.

For our analysis we need to associate information with each program point, that is, each labeled elementary block. For this, we introduce the notion of a *document type environment*: $\mathbf{DEnv} = \mathbf{DType}^{|\mathbf{DVar}|}$. Thus, a document type environment can be considered a map from the set **DVar** of document variables to the set of document types, **DType**.

Note that **GKind** and **FKind** are finite lattices. Since **GName**, **FName**, and **DVar** are all finite sets, both **GMap**, **FMap**, and **DEnv** inherit the property of being a finite lattice (with the obvious point-wise orderings).

3.4 Type Correctness

Our flow analysis will for each program point compute a document type environment that later is proven to soundly describe the possible values of document variables. The gap and field information is represented via a function solving the constraint set produced by the analysis:

$$A_{\text{entry}} : \mathbf{Lab} \rightarrow \mathbf{DEnv}.$$

Based on this function, we can determine if a program is type correct. The following overall condition ensures that only well-formed document values have been constructed: no gap or field map may contain the kind error. Furthermore, gap information is used to check plug operations:

- in $[x = x_1 < [y = x_2]]^l$, we must have $\pi_g(A_{\text{entry}}(l)(x_1))(y) = \mathsf{html}$, that is, when executing the operation, x_1 must have an HTML gap named y;
- in $[x = x' < [y = s]]^l$, we must have $\pi_g(A_{\text{entry}}(l)(x'))(y) \in \{\mathsf{string}, \mathsf{html}\}$, that is, x' must have a gap named y;

Finally, the field information imposes the following type requirement on the statement $[\texttt{show}\, x\, \texttt{receive}\, [v_1 = z_1, \ldots, v_n = z_n]]^l$:

$$typeof(v_i) \lessdot \pi_f(A_{\text{entry}}(l)(x))(z_i) \text{ for all } i.$$

The relation $\lessdot$ is defined in the section on semantics of statements.

Example The following example in <bigwig> syntax is type correct and builds an unbounded hierarchical structure that corresponds to the active threads in a bulletin board:

```
{ schema Message {int id; int re; string subject};
  relation Message Board;
  const html X = <html>
                   <li><[id]>:<[sub]>
                   <ul><[replies]></ul>
                 </html>;
  const html Y = <html><[reply]><[replies]></html>;

  html View(int id, string subject) {
    int i;
    html H = X<[id=id,sub=subject];
    vector Message R;
    R = sort(select * from Board where (#.re==id);
             id);
    for (i=0; i<|R|; i++)
      H = H<[replies =
                Y<[reply=View(R[i].id,R[i].subject)]
            ];
    return H;
  }

  show View(0,"");
}
```

This example cannot be generated using templates in MAWL.

4 Flow Analysis

The overall structure of our first-order forward data-flow analysis is quite standard. We generate a set of constraints for S_*. Through this system we define entry and exit environments for each labeled program point, relate them with monotone operations, and appeal to the general fixed-point theorem for finite lattices to obtain a minimal solution [10].

We present only an analysis for the simple imperative language DynDoc, but the techniques generalize in a straightforward manner to the monovariant interprocedural flow analysis that is required for the full <bigwig> language.

4.1 Terminology

In our type analysis we will be referring to the *initial label*, $init(S)$, of a statement S. The initial label of a sequence, $(S_1 ; S_2)$, is the initial label of S_1. The initial label of the other statement kinds is the unique label literally occurring in that statement.

Furthermore, we shall define for a statement, S, the set of *final labels*, $final(S)$. The final labels of a sequence are those of the second statement. The set of final labels for a conditional is the union of those of the two branches. The final label of a `while` statement is the label associated with the test. The final label of the other statement kinds is the unique label literally occurring in that statement.

Similarly, we define the set of *associated labels*, $labels(S)$. The set of labels associated with a sequence is just the union of the labels of the statements in the sequence. The labels associated with a conditional $\texttt{if}\, [b]^l\, S_1\, \texttt{else}\, S_2$ is the union of $\{l\}$, $labels(S_1)$, and $labels(S_2)$. For the while statement, $labels(\texttt{while}\, [b]^l\, S) = \{l\} \cup labels(S)$. The label associated with an assignment, a skip, and a show statement is just the label literally occurring in that statement. As mentioned earlier, we let **Lab** denote the set of labels associated with the program of interest, that is, $\mathbf{Lab} = labels(S_*)$.

Finally, we assume to have at our disposal the flow graph, G, of the program of interest S_*. We have $G = (\mathbf{Lab}, flow(S_*))$, where $flow(S) \subseteq \mathbf{Lab} \times \mathbf{Lab}$ is the set of flow edges associated with the statement S. More specifically, the set of flow edges associated with an assignment, a skip statement, and a show statement is simply the empty set. For sequences, conditionals, and while loops, we define $flow()$ as follows (see Figure 4):

$$\begin{aligned}
&flow(S_1 ; S_2) = \\
&\quad flow(S_1) \cup flow(S_2) \cup \{\, (l, init(S_2)) \mid l \in final(S_1) \,\} \\
&flow(\texttt{if}\, [b]^l\, S_1\, \texttt{else}\, S_2) = \\
&\quad flow(S_1) \cup flow(S_2) \cup \{(l, init(S_1)), (l, init(S_2))\} \\
&flow(\texttt{while}\, [b]^l\, S) = \\
&\quad flow(S) \cup \{(l, init(S))\} \cup \{\, (l', l) \mid l' \in final(S) \,\}
\end{aligned}$$

Since we will be defining a forward analysis, we shall make the simplifying assumption that the program of interest S_* has an *isolated entry*, that is, $init(S_*)$ has no incoming flow-edges. This can be ensured by prepending S_* with a `skip` statement.

4.2 The analysis

We introduce a constraint system via the functions $\mathsf{A}_{\text{entry}}$ and A_{exit} that map labels to environments, that is:

$$\mathsf{A}_{\text{entry}}, \mathsf{A}_{\text{exit}} : \mathbf{Lab} \to \mathbf{DEnv}$$

For $l = init(S_*)$, we require that $\mathsf{A}_{\text{entry}}(l)$ corresponds to the start state σ_0 of S_*. That is, for any x we have $\mathsf{A}_{\text{entry}}(l)(x) \sqsupseteq (gap \otimes field)(\sigma_0(x))$. For all other l, $\mathsf{A}_{\text{entry}}(l)$ must be greater than or equal to each of the exit environments from incoming flow-edges, that is:

$$\mathsf{A}_{\text{entry}}(l) \sqsupseteq \bigsqcup \{\, \mathsf{A}_{\text{exit}}(l') \mid (l', l) \in flow(S_*) \,\}$$

For labels, l, associated with `skip` statements, conditionals, `while` loops, or `show` calls, we simply require $\mathsf{A}_{\text{exit}}(l) \sqsupseteq \mathsf{A}_{\text{entry}}(l)$. If a label l is associated with an assignment $[x = d]^l$, we impose:

$$\mathsf{A}_{\text{exit}}(l) \sqsupseteq \mathsf{A}_{\text{entry}}(l)[x \mapsto \mathrm{A}[\![d]\!](\mathsf{A}_{\text{entry}}(l))]$$

where the map $\mathrm{A}[\![\cdot]\!] : \mathbf{DExp} \to \mathbf{DEnv} \to \mathbf{DType}$ is defined as follows (using the function notation):

$$\begin{aligned}
\mathrm{A}[\![x]\!]denv &= denv(x) \\
\mathrm{A}[\![const]\!]denv &= (gap \otimes field)(const) \\
\mathrm{A}[\![x_1 < [y = x_2]]\!]denv &= denv(x_1)[y \mapsto \mathsf{nogap}] \oplus denv(x_2) \\
\mathrm{A}[\![x < [y = s]]\!]denv &= denv(x)[y \mapsto \mathsf{nogap}]
\end{aligned}$$

The notation $dt[y \mapsto \mathsf{nogap}]$ simply means the document type that acts as dt except for the gap map part which maps x to nogap.

The monotone operators $\oplus_g$ and $\oplus_f$ are extended from the previous definitions to **GKind** and **FKind**, respectively, as follows:

$$a \oplus_g b = \begin{cases} \mathsf{error} & \text{if } a,b \in \{\mathsf{string}, \mathsf{html}\} \\ a & \text{if } b = \mathsf{nogap} \\ b & \text{if } a = \mathsf{nogap} \\ a \sqcup b & \text{otherwise} \end{cases}$$

$$a \oplus_f b = \begin{cases} \mathsf{error} & \text{if } a = b = \mathsf{ordinary} \\ \mathsf{rel}(F) & \text{if } a = b = \mathsf{tuple}(F) \\ a & \text{if } b = \mathsf{nofield} \\ b & \text{if } a = \mathsf{nofield} \\ a \sqcup b & \text{otherwise} \end{cases}$$

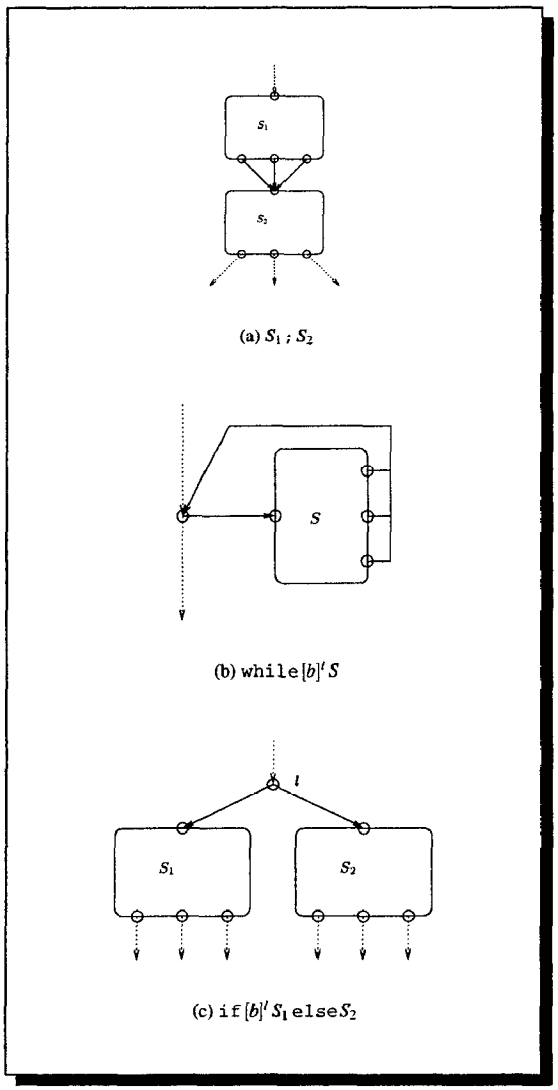


Figure 4: Flow of sequencing, while statements, and conditionals.

```
for all l ∈ Lab {A_entry(l) = A_exit(l) = ⊥_DEnv};
W = Lab;
while (W ≠ ∅) {
    l = remove_one(W);
    new_entry = ⊔{A_entry(l') | (l',l) ∈ flow(S*)};
    if (new_entry ≠ A_entry(l)) {
        A_entry(l) = new_entry;
        new_exit = F_l(new_entry);
        if (new_exit ≠ A_exit(l)) {
            A_exit(l) = new_exit;
            W = W ∪ {l' | (l,l') ∈ flow(S*)};
        }
    }
}
```

Figure 5: The worklist algorithm.

They can easily be lifted to **GMap** and **FMap**, respectively. For **DType**, we define

$$(g_1, f_1) \oplus (g_2, f_2) = (g_1 \oplus_g g_2, f_1 \oplus_f f_2).$$

4.3 Solving the constraint system efficiently

Since we are only working with finite lattices and all our transfer functions are monotone, we can use standard techniques to find the smallest solution to the constraints generated from the program S_*. In this section, we describe how the so-called worklist algorithm is used to compute the smallest solution in terms of the least fixed point of a monotone function. Then we present an optimization that reduces the size of the flow graph substantially and thus the time needed to compute a solution. The optimization is successful mainly because we are performing a domain-specific analysis leaving large parts of the program and thus large parts of the flow-graph irrelevant for the analysis.

Classical fixed point computation In general we consider the two functions A_{entry} and A_{exit} as elements in **Lab** $\rightarrow$ **DEnv**. If substituting the variables $\mathsf{A}_{\text{entry}}$ and A_{exit} with the elements A_{entry} and A_{exit} makes the constraint set true, that is, if $A \models \mathsf{A}^{\sqsubseteq}(S_*)$, we have found a solution. What we want is the smallest such solution. The smallest solution is found simply by considering the generated constraints as the definition of a monotone function, F, from the finite lattice (**Lab** $\rightarrow$ **DEnv**) $\times$ (**Lab** $\rightarrow$ **DEnv**) to itself. Starting with the bottom element, that is, the pair consisting of two functions that both map every label to the bottom element of **DEnv**, we can iterate using F and find the smallest fixed point as $\bigsqcup F^n(\bot)$. By monotonicity, we can show by induction that $F^n(\bot) \sqsubseteq F^{n+1}(\bot)$. Thus, by finiteness the our lattice we have a safe way of computing the smallest solution to our constraint set. More specifically, we find the fixed point using the following worklist algorithm in Figure 4.3.

Flow-graph reduction Due to the fact that our analysis is concerned only with document template computations, we can collapse the flow-graph substantially. More, specifically we make use of the four sound flow-graph transformation rules in Figure 6, where a single circled node indicates that the node could be associated with any program point and a double circled node denotes an "identity

node", that is, a node in which the entry and exit maps are identical. Using these four rules as long as they apply will as we indicate through a number of examples below reduce the size of the flow-graph considerably. An efficient strategy for using these four rules is shown in Figure 4.3 (where $V = \mathbf{Lab}$ and $E = \mathit{flow}(S_*)$). That

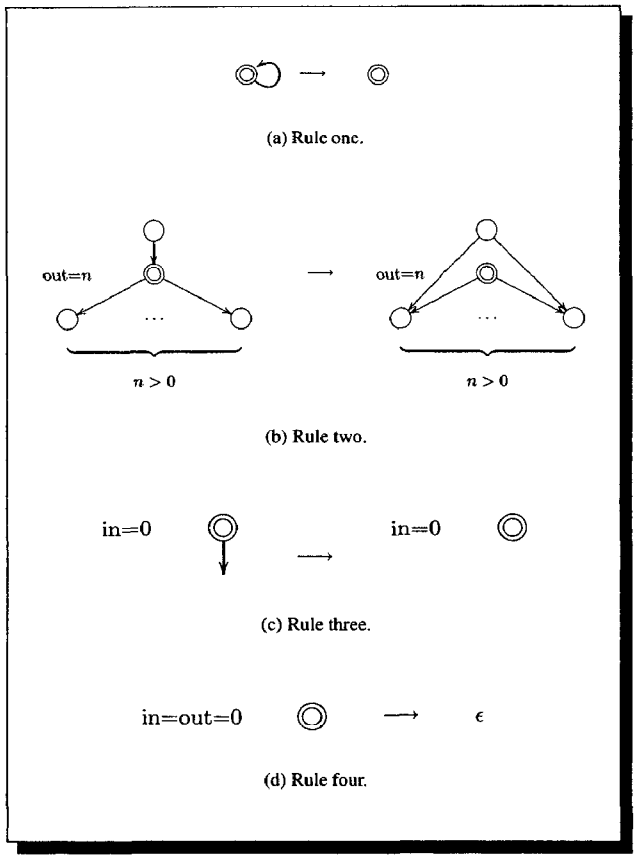


Figure 6: Flow-graph transformation rules.

is, we consider the nodes in the graph one by one and then for each identity node apply rule one if possible and then if the out degree of the node is positive, we iteratively use rule two until the in degree of the node is zero. After this rule three is used repeatedly until the out degree reaches zero, and we can finally use rule four to remove the by now possibly redundant identity node. We then move on to the next identity node.

Using the sound graph reduction technique presented above we can reduce the size of the flow-graph to be analyzed by a factor of three allowing for a speedup of the analysis of roughly 50% as indicated by the diagrams in Figure 8(a) and 8(b) for the test programs `appointment`, `wwwboard`, `calendar`, `cdshop`, and `tournament`.

4.4 Correctness of the analysis

In this section we shall prove that the gap and field analysis defined previously in fact safely collects information about the gaps and fields occurring during computation. Before proving the correctness results we shall establish the following property of the semantics:

Lemma 4.1 *If* $\langle S,\sigma\rangle \rightarrow \langle S',\sigma'\rangle$ *then* $\mathit{final}(S) \supseteq \mathit{final}(S')$, $\mathit{flow}(S) \supseteq \mathit{flow}(S')$, *and* $\mathit{labels}(S) \supseteq \mathit{labels}(S')$.

```
for all id nodes l ∈ V {
   if ((l,l) ∈ E) apply rule one;
   if ({l' | (l,l') ∈ E} ≠ ∅) {
      while ({l' | (l',l) ∈ E} ≠ ∅)
          apply rule two;
      while ({l' | (l,l') ∈ E} ≠ ∅)
          apply rule three;
   }
   if ({l' | (l,l') ∈ E ∨ (l,l') ∈ E} = ∅)
          apply rule four;
}
```

Figure 7: The reduction algorithm.

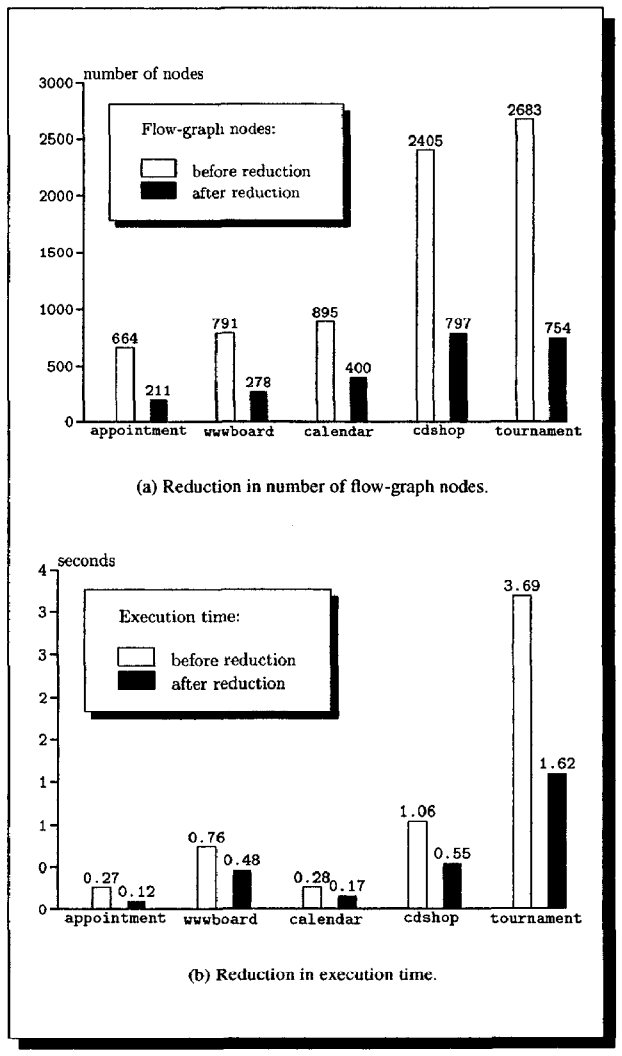


Figure 8: Effect of flow-graph reductions.

Proof: (A similar proof appears in [10]). First, let us show that $final(S) \supseteq final(S')$. The proof is by case analysis on the semantic rule used to establish $\langle S,\sigma\rangle \rightarrow \langle S',\sigma'\rangle$. Consulting Figure 1, we see that there are five non-vacuous cases:

The case $[seq_1]$: Assuming $\langle S_1 ; S_2,\sigma\rangle \rightarrow \langle S_1' ; S_2,\sigma'\rangle$ we get:

$$final(S_1 ; S_2) = final(S_2) = final(S_1' ; S_2).$$

The case $[seq_2]$: Then $\langle S_1 ; S_2,\sigma\rangle \rightarrow \langle S_2,\sigma'\rangle$ and we get:

$$final(S_1 ; S_2) = final(S_2).$$

The case $[if_1]$: Thus $\langle \texttt{if}\,[b]^l\, S_1\, \texttt{else}\, S_2,\sigma\rangle \rightarrow \langle S_1,\sigma\rangle$ and we get:

$$final(\texttt{if}\,[b]^l\, S_1\, \texttt{else}\, S_2) = final(S_1) \cup final(S_2) \supseteq final(S_1).$$

The case $[if_2]$ is similar to the case $[if_1]$.

The case $[wh_1]$: Since $\langle \texttt{while}\,[b]^l\, S,\sigma\rangle \rightarrow \langle (S;\, \texttt{while}\,[b]^l\, S),\sigma\rangle$ we get that:

$$final(S;\, \texttt{while}\,[b]^l\, S) = final(\texttt{while}\,[b]^l\, S).$$

This completes the proof that $final(S) \supseteq final(S')$.

Now, let us turn to the proof that $flow(S) \supseteq flow(S')$ which is established by induction on the shape of the inference tree used to establish $\langle S,\sigma\rangle \rightarrow \langle S',\sigma'\rangle$. Again, there are five non-vacuous cases:

The case $[seq_1]$: Assuming $\langle S_1 ; S_2,\sigma\rangle \rightarrow \langle S_1' ; S_2,\sigma'\rangle$ because $\langle S_1,\sigma\rangle \rightarrow \langle S_1',\sigma'\rangle$ we get (using the induction hypothesis and $final(S_1) \supseteq final(S_1')$):

$$\begin{aligned} flow(S_1 ; S_2) &= flow(S_1) \cup flow(S_2) \cup \{ (l, init(S_2)) \mid l \in final(S_1) \} \\ &\supseteq flow(S_1') \cup flow(S_2) \cup \{ (l, init(S_2)) \mid l \in final(S_1) \} \\ &\supseteq flow(S_1') \cup flow(S_2) \cup \{ (l, init(S_2)) \mid l \in final(S_1') \} \\ &= flow(S_1' ; S_2). \end{aligned}$$

The case $[seq_2]$: Then $\langle S_1 ; S_2,\sigma\rangle \rightarrow \langle S_2,\sigma'\rangle$ because $\langle S_1,\sigma\rangle \rightarrow \sigma'$ and we get:

$$\begin{aligned} flow(S_1 ; S_2) &= flow(S_1) \cup flow(S_2) \cup \{ (l, init(S_2)) \mid l \in final(S_1) \} \\ &\supseteq flow(S_2). \end{aligned}$$

The case $[if_1]$: Thus $\langle \texttt{if}\,[b]^l\, S_1\, \texttt{else}\, S_2,\sigma\rangle \rightarrow \langle S_1,\sigma\rangle$ and we get:

$$\begin{aligned} flow(S_1 ; S_2) &= flow(S_1) \cup flow(S_2) \cup \{(l, init(S_1)), (l, init(S_2))\} \\ &\supseteq flow(S_1). \end{aligned}$$

The case $[if_2]$ is similar to the case $[if_1]$.

The case $[wh_1]$: Since $\langle \texttt{while}\,[b]^l\, S,\sigma\rangle \rightarrow \langle (S;\, \texttt{while}\,[b]^l\, S),\sigma\rangle$ we get that:

$$\begin{aligned} &flow(S;\, \texttt{while}\,[b]^l\, S) \\ &\quad = flow(S) \cup flow(\texttt{while}\,[b]^l\, S) \cup \{ (l', l) \mid l' \in final(S) \} \\ &\quad = flow(S) \cup (flow(S) \cup \{(l, init(S))\} \cup \{ (l', l) \mid l' \in final(S) \}) \\ &\qquad \cup \{ (l', l) \mid l' \in final(S) \} \\ &\quad = flow(S) \cup \{(l, init(S))\} \cup \{ (l', l) \mid l' \in final(S) \} \\ &\quad = flow(\texttt{while}\,[b]^l\, S). \end{aligned}$$

This completes the proof that $flow(S) \supseteq flow(S')$.

The proof of $labels(S) \supseteq labels(S')$ is similar to that of $flow(S) \supseteq flow(S')$ and we thus omit the details. ∎

Previously we showed how to define a gap and field constraint system for a labeled program S_*; we will refer to this system as $\mathsf{A}^{\sqsubseteq}(S_*)$. Given functions $A_{\text{entry}}, A_{\text{exit}}$, we say that A solves $\mathsf{A}^{\sqsubseteq}(S_*)$, and write $A \models \mathsf{A}^{\sqsubseteq}(S_*)$ if the functions satisfy the constraints.

The next lemma shows that if we have a solution to the constraint system corresponding to some statement S, then it will also be a solution to the constraint system obtained from a sub-statement S'; this result will be essential in the proof of the correctness result.

Lemma 4.2 *If $flow(S) \supseteq flow(S')$, $labels(S) \supseteq labels(S')$, and $A \models \mathsf{A}^{\sqsubseteq}(S)$ then $A \models \mathsf{A}^{\sqsubseteq}(S')$.*

Proof: Assume $flow(S) \supseteq flow(S')$, $labels(S) \supseteq labels(S')$, and $A \models \mathsf{A}^{\sqsubseteq}(S)$. The result in the lemma then follows from the observation that if $A \models \mathsf{A}^{\sqsubseteq}(S)$, that is, if A_{entry} and A_{exit} satisfy the constraints in $\mathsf{A}^{\sqsubseteq}(S)$, then they must also satisfy the smaller set of constraints, $\mathsf{A}^{\sqsubseteq}(S')$. Thus $A \models \mathsf{A}^{\sqsubseteq}(S')$ as was needed. ∎

We now have the following corollary expressing that a solution to the constraints of $\mathsf{A}^{\sqsubseteq}$ are preserved during computation.

Corollary 4.3 *If $\langle S,\sigma\rangle \rightarrow \langle S',\sigma'\rangle$ and $A \models \mathsf{A}^{\sqsubseteq}(S)$ then $A \models \mathsf{A}^{\sqsubseteq}(S')$.*

Proof: The corollary follows from Lemma 4.1 and 4.2. ∎

We shall define the correctness relation, $\lesssim \subseteq \mathbf{State} \times \mathbf{DEnv}$, as follows:

$$\sigma \lesssim denv \text{ if and only if for all } x : (gap \otimes field)(\sigma(x)) \sqsubseteq denv(x).$$

Notice, that if $denv \sqsubseteq denv'$ then by definition of $\lesssim$, $\sigma \lesssim denv$ implies $\sigma \lesssim denv'$.

We have the following lemma expressing that the D and A functions preserve the property of being related by the correctness relation.

Lemma 4.4 *If $\sigma \lesssim denv$ and $\mathrm{D}[\![d]\!]\sigma$ is defined then*

$$(gap \otimes field)(\mathrm{D}[\![d]\!]\sigma) \sqsubseteq \mathrm{A}[\![d]\!]denv.$$

Proof: The proof is by case analysis on the document expression kind. Thus assume $\sigma \lesssim denv$ and $\mathrm{D}[\![d]\!]\sigma$ is defined. There are four cases to consider:

The case $d = x$: From $\sigma \lesssim denv$ we get $(gap \otimes field)(\sigma(x)) \sqsubseteq denv(x)$ for all x. Now, since $\mathrm{D}[\![x]\!]\sigma = \sigma(x)$ and $\mathrm{A}[\![x]\!]denv = denv(x)$ we get:

$$(gap \otimes field)(\mathrm{D}[\![d]\!]\sigma) \sqsubseteq \mathrm{A}[\![d]\!]denv.$$

The case $d = const$: Since $\mathrm{D}[\![const]\!]\sigma = const$ and $\mathrm{A}[\![const]\!]denv = (gap \otimes field)(const)$, we immediately get that:

$$(gap \otimes field)(\mathrm{D}[\![const]\!]\sigma) = (gap \otimes field)(const) = \mathrm{A}[\![const]\!]denv.$$

The case $d = (x_1 < [\, y = x_2 \,])$: We get:

$$\begin{aligned} &(gap \otimes field)(\mathrm{D}[\![d]\!]\sigma) \\ &\quad = (gap \otimes field)(\mathrm{D}[\![x_1 < [\, y = x_2 \,]]\!]\sigma) \\ &\quad = (gap \otimes field)(plug(\sigma(x_1), y, \sigma(x_2))) \\ &\quad = ((gap \otimes field)(\sigma(x_1)))[y \mapsto \mathsf{nogap}] \oplus (gap \otimes field)(\sigma(x_2)) \\ &\quad \sqsubseteq denv(x_1)[y \mapsto \mathsf{nogap}] \oplus denv(x_2) \\ &\quad = \mathrm{A}[\![x_1 < [\, y = x_2 \,]]\!]denv \\ &\quad = \mathrm{A}[\![d]\!]denv \end{aligned}$$

where the second line follows from the assumption that $\mathbf{D}[\![d]\!]\sigma$ is defined, third line from the fact that:

$$gap(plug(c_1,y,c_2)) = gap(c_1)[y \mapsto \mathsf{nogap}] \oplus_g gap(c_2) \text{ and}$$
$$field(plug(c_1,y,c_2)) = field(c_1) \oplus_f field(c_2)$$

if $plug(c_1,y,c_2)$ is defined, and the fourth line from the assumption $\sigma \lesssim denv$.

The case $d = (x_1 < [y = s])$ is similar to (but simpler than) the case $d = (x_1 < [y = x_2])$ and is thus omitted.

This completes the proof that $\sigma \lesssim denv$ implies $(gap \otimes field)(\mathbf{D}[\![d]\!]\sigma) \sqsubseteq \mathbf{A}[\![d]\!]denv$. ∎

We are now ready for the main result. It states how semantically correct information is preserved under each step of the execution.

Theorem 4.5 *Assume $A \models \mathsf{A}^{\sqsubseteq}(S)$ and $\sigma \lesssim A_{entry}(init(S))$.*

- *If $\langle S,\sigma\rangle \to \langle S',\sigma'\rangle$ then $\sigma' \lesssim A_{entry}(init(S'))$.*
- *If $\langle S,\sigma\rangle \to \sigma'$ then $\sigma' \lesssim \bigsqcup\{A_{exit}(l) \mid l \in final(S)\}$.*

Proof: The proof is by induction on the shape of the inference tree used to establish $\langle S,\sigma\rangle \to \langle S',\sigma'\rangle$ and $\langle S,\sigma\rangle \to \sigma'$, respectively. Thus assume $A \models \mathsf{A}^{\sqsubseteq}(S)$ and $\sigma \lesssim A_{\text{entry}}(init(S))$. Consulting Figure 1 we see that there are nine cases to consider.

The case $[asn]$: Thus $\langle [x = d]^l,\sigma\rangle \to \sigma[x \mapsto \mathbf{D}[\![d]\!]\sigma]$. What we need to show is that $\sigma' \lesssim A_{\text{exit}}(l)$, where $\sigma' = \sigma[x \mapsto \mathbf{D}[\![d]\!]\sigma]$. Since $A \models \mathsf{A}^{\sqsubseteq}([x = d]^l)$, we get $A_{\text{exit}}(l) \sqsupseteq A_{\text{entry}}(l)[x \mapsto \mathbf{A}[\![d]\!](A_{\text{entry}}(l))]$. Thus it is enough to show $(gap \otimes field)(\mathbf{D}[\![d]\!]\sigma) \sqsubseteq \mathbf{A}[\![d]\!](A_{\text{entry}}(l))$. But this follows from Lemma 4.4 since $\sigma \lesssim A_{\text{entry}}(l)$. We conclude that $\sigma' \lesssim A_{\text{exit}}(l)$.

The case $[skip]$: Then $\langle [\mathtt{skip}]^l,\sigma\rangle \to \sigma$. Since $A \models \mathsf{A}^{\sqsubseteq}([\mathtt{skip}]^l)$ we get that $A_{\text{exit}}(l) \sqsupseteq A_{\text{entry}}(l)$. From the fact that $final([\mathtt{skip}]^l) = \{l\}$ and the assumption $\sigma \lesssim A_{\text{entry}}(l)$ we then conclude that $\sigma \lesssim \bigsqcup\{A_{\text{exit}}(l) \mid l \in final(S)\}$ as needed.

The case $[seq_1]$: Then $\langle S_1 ; S_2,\sigma\rangle \to \langle S_1' ; S_2,\sigma'\rangle$ because $\langle S_1,\sigma\rangle \to \langle S_1',\sigma'\rangle$. From the assumption $A \models \mathsf{A}^{\sqsubseteq}(S_1 ; S_2)$ and the fact that $flow(S_1 ; S_2) \supseteq flow(S_1)$, it follows using Lemma 4.2 that $A \models \mathsf{A}^{\sqsubseteq}(S_1)$. By assumption,

$$\sigma \lesssim A_{\text{entry}}(init(S_1 ; S_2)) = A_{\text{entry}}(init(S_1)).$$

Thus using the induction hypothesis we get

$$\sigma' \lesssim A_{\text{entry}}(init(S_1')) = A_{\text{entry}}(init(S_1' ; S_2))$$

as needed.

The case $[seq_2]$: Then $\langle S_1 ; S_2,\sigma\rangle \to \langle S_2,\sigma'\rangle$ because $\langle S_1,\sigma\rangle \to \sigma'$. Again, using the induction hypothesis we get:

$$\sigma' \lesssim \bigsqcup\{A_{\text{exit}}(l) \mid l \in final(S_1)\}.$$

Since $\{(l, init(S_2)) \mid l \in final(S_1)\} \subseteq flow(S_1 ; S_2)$ and $A \models \mathsf{A}^{\sqsubseteq}(S_1 ; S_2)$ we get $A_{\text{entry}}(init(S_2)) \sqsupseteq \bigsqcup\{A_{\text{exit}}(l) \mid l \in final(S_1)\}$. We conclude that $\sigma' \lesssim A_{\text{entry}}(init(S_2))$ as needed.

The case $[if_1]$: Then $\langle \mathtt{if}\,[b]^l\,S_1\,\mathtt{else}\,S_2,\sigma\rangle \to \langle S_1,\sigma\rangle$. Since $A \models \mathsf{A}^{\sqsubseteq}(\mathtt{if}\,[b]^l\,S_1\,\mathtt{else}\,S_2)$ and $(i, init(S_1)) \in flow(\mathtt{if}\,[b]^l\,S_1\,\mathtt{else}\,S_2)$ we get $A_{\text{entry}}(l) \sqsubseteq A_{\text{entry}}(init(S_1))$. Thus, $\sigma \lesssim A_{\text{entry}}(init(S_1))$ as needed.

The case $[if_2]$ is similar to the case $[if_1]$ and is thus omitted.

The case $[wh_1]$: Then $\langle \mathtt{while}\,[b]^l\,S,\sigma\rangle \to \langle (S; \mathtt{while}\,[b]^l\,S),\sigma\rangle$. Since A is a solution to the constraint set and $(l, init(S)) \in flow(\mathtt{while}\,[b]^l\,S)$, we get that $A_{\text{entry}}(init(S)) \sqsupseteq A_{\text{exit}}(l) \sqsupseteq A_{\text{entry}}(l)$. Thus since by assumption $\sigma \lesssim A_{\text{entry}}(l)$, we get $\sigma \lesssim A_{\text{entry}}(init(S)) = A_{\text{entry}}(init(S; \mathtt{while}\,[b]^l\,S))$.

The cases $[wh_2]$ and $[show]$ are similar to the case $[skip]$ and are thus omitted.

This completes the proof. ∎

Using $\sigma_0 \lesssim A_{\text{entry}}(init(S_*))$ for the base case and Lemma 4.3 for the induction step, simple induction on the length of the derivation sequences

$$\langle S,\sigma_0\rangle \to^* \langle S',\sigma'\rangle \text{ and } \langle S,\sigma_0\rangle \to^* \sigma'$$

will extend the above result to arbitrary computation sequences.

Using the correctness result for the analysis, we can show that type-checked programs never encounter runtime errors:

Corollary 4.6 *Assume S_* type checks. Then the execution of S_* will never enter a stuck configuration.*

Proof: The result follows from the correctness of the gap and field analysis and the close relation between type correctness and semantics of DynDoc.

Assume that our program of interest, S_* has been analyzed for gaps and fields and based on that checked to be well-typed. We show that during execution of S_* we will never enter a stuck configuration. Assume on the contrary, that we have reached a stuck configuration. By inspecting the semantics of statements, we see that this can either be due to an assignment in which an illegal plug operation is performed or due to a show call in which the form fields of the shown document do not match the types of the receiving program variables.

In the former case $([x = x_1 < [y = x_2]]^l)$ there are two possible reasons why the plug operation is illegal: either there is no y gap in x_1 or the resulting document is not well-formed. We can rule out the first immediately since type correctness requires $\pi_g(A_{\text{entry}}(l)(x_1))(y) = \mathsf{html}$ and by correctness of the analysis, x_1 *must* have an HTML gap named y in the current state. The second we can rule out by the following argument: since the program type-checked, there are no occurrences of the error kind in the analysis. More specifically, $A_{\text{exit}}(l)$ does not have any error kinds associated. Thus, by correctness of the analysis and the close relation between semantics of the plug operation and the definition of the analysis, the resulting document must be well-formed.

Now, in the case of the `show-receive` call, we can rule out the possibility of a mismatch between the form fields of the shown document and the types of the receiving program variables by referring to the correctness of the analysis and the direct connection between the semantics of show-receive and the definition of type correctness that both use the $\lessdot$ relation. ∎

Experience has shown that almost all correct programs do in fact type-check. That is, our analysis is not only sound but also precise enough to be useful in a practical setting.

4.5 Extensions

In this section we treat the implemented extension to full monovariant interprocedural flow analysis and type-checking, and we describe how a slight change in the ordering on gap kinds along with implicit gap closing can conveniently be used to extend the set of programs accepted by our analysis.

Interprocedural analysis In our implementation of the presented type system, the flow analysis is extended in a standard way to a full monovariant *interprocedural* first-order forward data-flow analysis. For simplicity, we have presented the analysis and correctness results for the core language only, but the results of course extend to the full interprocedural analysis.

Implicit gap closing If we change the ordering on **GKind** slightly then the compiler can implicitly close gaps occurring in some but not all incoming flow edges to a given program point. This implicit *coercion* allows for more flexibility in the programming of Web services without sacrificing the important possibility of checking static safety. The implicit closing described in this section is part of our implementation.

As mentioned, we change the ordering on **GKind** slightly. More precisely, the ordering is changed to the total ordering:

$$\bot \sqsubseteq \mathsf{error} \sqsubseteq \mathsf{html} \sqsubseteq \mathsf{string} \sqsubseteq \mathsf{nogap}.$$

By examination of the transfer function for assignments, it is clear that the transfer function is still monotone and thus that the flow analysis is still well-defined. This slight change will allow for more programs to type-check. For instance, the following code fragment would not type-check in the original type system:

```
string name, title, univ;
html Intro = <html>
                <[title]><[name]><[affil]>
             </html>;
html Affil = <html> Ph.D. from <[univ]> </html>;
...collect name, title, etc. ...
if (title=="Doctor")
   Intro = Intro<[affil=(Affil<[univ=univ])];
show Intro<[name=name,title=title];
```

The code will not type-check because after the conditional, `Intro` might or might not have an `affil` gap. This problem could of course be solved simply by changing the conditional to

```
if (title=="Doctor")
   Intro = Intro<[affil=(Affil<[univ=univ])]
else Intro = Intro<[affil=""];
```

However, experience has shown that the drag of dealing with such technicalities becomes rather annoying. With our new ordering on **GKind**, the above code fragment would in fact type-check. At the end of the conditional, the least upper bound of incoming flow-edges would instead of an error result in a nogap for the gap name `affil` in the document variable `Intro`.

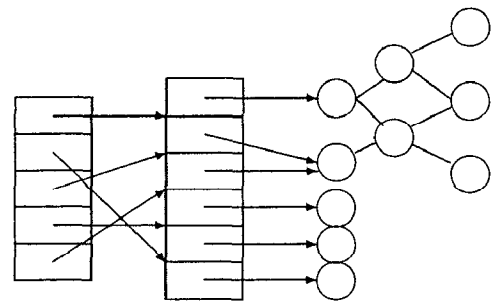
The crucial observation then, is that a program which type-checks with the new ordering can by means of the information inferred by the analysis easily be transformed into a program which type-checks in the original system and which has the expected behavior.

Note that the only new things that can be accepted with the changed ordering are merging of flow-edges in which for one or more gaps, some incoming flow edges deliver gaps and some do not. In the old system this would give rise to the top element error. Also, this observation explains how the program transformation should change programs to make them type check in the original type system: for each node in which the gap maps of incoming flow edges differ, insert explicit gap closing along the appropriate paths such that in fact all incoming flow edges will have the same gaps. The transformed program will then type-check with the old ordering on **GKind**. Thus relying on our correctness results we can still statically check the program for errors in document computations, that is, we can provide static safety if the program is well-typed.

5 Runtime Implementation

A naive implementation of dynamic documents would at runtime represent each document as a complete parse tree. A document would then require memory proportional to its printed size, and plugging a document of size n into one of size m would require time $O(n+m)$. For well-typed programs, we can do much better than that: documents are represented with almost complete sharing, and the plug operation only takes time $O(1)$ while preserving efficiency of the show operation.

The key point it to represent a document value as a *closure*, which is essentially a table of function calls. The general structure of a closure is as follows:



switchboard contents call dag

In a document with k gaps, the *switchboard* has k entries and the *contents* table has $k+1$ entries. The switchboard is a map from gap names in lexicographic order to the order in which they occur in the document. The contents table defines the contents between gaps by pointing to nodes in a *call dag* (directed acyclic graph). Each node in the call dag has a reference counter. Internal nodes branch in two, while each leaf consists of either a string value or a function pointer with an integer argument. To be quite explicit, the corresponding type declaration in C is:

```
typedef struct dynDoc {
  int size;
  int *switchboard;
  struct callDag **contents;
} dynDoc;

typedef struct callDag {
  int refcount;
  enum {stringK,funcK,nodeK} kind;
  union {
    char *stringD;
    struct {void (*p)(int); int arg;} funcD;
    struct {struct callDag *left,*right;} nodeD;
  } val;
}
```

The reference count in the call dag nodes is used for automatic garbage collection (there are no cycles in the representation). The switchboard is required since document types do not specify the order of gaps, and two flow edges may choose different permutations. For instance, in a page with addresses, the appearance of street name and number would be different in a US page and a Continental European page.

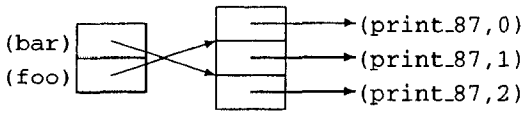
For each document constant we construct a function which on input i prints the contents between gaps number $i-1$ and i. For the fragment:

```
This <[foo]> <b>is <[bar]></b> you know!
```

the constructed function could be:

```
void print_87(int i) {
  switch (i) {
    case 0: printf("This"); break;
    case 1: printf("<b>is"); break;
    case 2: printf("</b> you know!"); break;
    }
}
```

where the index 87 is some identification of the constant fragment. The corresponding document value has the obvious switchboard and the i'th entry in its contents table is a function node with function pointer `print_87` and argument i. For the example above, the representation is:



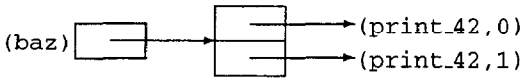
The textual contents of a closure can be printed by a simple in-order traversal of the dag which for each leaf node either calls the given function with the given argument or prints the given string constant. Because of the dag structure, the textual contents may be exponentially larger than the runtime representation. Note that this printing technique implicitly plugs every remaining gap with the empty document or string. The plug operation is realized by a function:

```
dynDoc *docPlugHTML(dynDoc *a, int i,
                    dynDoc *b, int *merge)
```

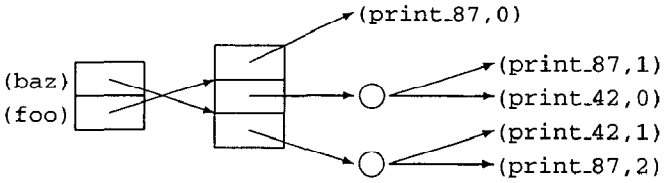
which plugs `b` into the HTML gap in `a` whose lexicographic index is `i`. Note that the value of `i` can be determined from the statically computed document type of `a`. The created document will have array size `(a->size)+(b->size)-1`, so its two tables are allocated accordingly.

The contents table is constructed as follows. Assume that the switchboard of `a` sends `i` to point between entries j and $j+1$ in the contents table. The contents table of `a` is cut in two after entry j. The contents table of `b` is inserted in between. At the two joints the neighboring elements are combined by creating new internal dag nodes, which leaves a contents table of the correct length.

The switchboard table is now created, using the `merge` argument which for every entry in the switchboard table of `b` indicates the position in the the switchboard table of `a` after which it must be merged to preserve the lexicographical order. Note that `merge` can be computed from the static document types of `a` and `b`. If the switchboard of `a` contains the gap names `(a,b,e,f)` and that of `b` contains `(c,d,g)` then the `merge` list would be `(2,2,4)`. If in the above example document we plug the gap named `bar` with the document:



then the result becomes:



Since the maximal length of a switchboard is a constant determined from the program text, the plug operation executes in time $O(1)$. A similar, but simpler, function is defined for plugging string gaps.

6 Conclusions

Interactive Web services based on the CGI protocol and HTML forms can be made safe, convenient, and efficient by using the typed higher-order document templates that we propose. An efficient runtime implementation is also provided.

Acknowledgment: The entire <bigwig> team has assisted in implementing and experimenting with typed dynamic documents.

References

[1] David Atkins, Thomas Ball, Michael Benedikt, Glenn Bruns, Kenneth Cox, Peter Mataga, and Kenneth Rehor. Experience with a domain specific language for form-based services. In *Usenix Conference on Domain Specific Languages*, Santa Barbara, CA, October 1997.

[2] Leon Atkinson. *Core PHP Programming*. Prentice Hall Computer Books, May 1999.

[3] Claus Brabrand, Anders Møller, Anders Sandholm, and Michael I. Schwartzbach. <bigwig> — a language for developing interactive Web services. In preparation, 1999.

[4] Alex Fedorov, Richard Harrison, Dave Sussman, Brian Francis, and Stephen Wood. *Professional Active Server Pages 2.0*. Wrox Press Inc., 2nd edition, March 1998.

[5] Shishir Gundavaram. *CGI Programming on the World Wide Web*. O'Reilly & Associates, Inc., 1996.

[6] Jason Hunter, William Crawford, and Paula Ferguson. *Java Servlet Programming*. O'Reilly & Associates, December 1998.

[7] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall International series in computer science. Prentice-Hall, 1987.

[8] D. A. Ladd and J. C. Ramming. Programming the Web: An application-oriented language for hypermedia services. In *4th Intl. World Wide Web Conference*, 1995.

[9] Michael R. Levy. Web programming in Guide. *Software—Practice and Experience*, 28(15):1581–1603, December 1998.

[10] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis: Flows and Effects*. Springer, 1999.

[11] Larry Wall, Tom Christensen, and Randal L. Schwartz. *Programming Perl*. O'Reilly & Associates, Inc., second edition edition, September 1996.